Institute for Parallel and Distributed Systems - Scientific Computing
University of Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Bachelor Thesis

Parameter-dependent self-learning optimization

Tareq Abu El Komboz

Course of Study:	Informatik
Examiner:	Prof. Dr. rer. nat. Dirk Pflüger
Supervisor:	Peter Domanski, M.Sc.
Commenced:	April 5, 2022
Completed:	November 2, 2022

Abstract

Manually developing optimization algorithms is a time-consuming task requiring expert knowledge. Therefore, it makes a lot of sense to automate the design process of such algorithms. Additionally, learned optimization algorithms reduce the number of a priori assumptions made about the characteristics of the underlying objective function. Numerous works discuss possibilities for learning optimization algorithms. This field of study is called learn-to-optimize. In this bachelor's thesis, we concentrate on the reinforcement learning perspective. Consequently, optimization algorithms are represented as policies. The comparison of learned algorithms to current state-of-the-art algorithms for particular applications reveals that learned algorithms manage to perform better concerning convergence speed and final objective function value. However, most existing approaches only consider fixed sets of parameters to be optimized. Because of this, it is challenging to adapt the learned optimization algorithm to other objective functions. More importantly, it is impossible to optimize when explicit constraints on so-called "free" optimization parameters are given. We investigated the learn-to-optimize approach under various optimization parameter sets and conditions on "free" parameters to solve this problem. Furthermore, we studied the performance of learned optimizers in high-dimensional setups.

Contents

List of Figures

1 Introduction

Optimization problems find application in every essential scientific domain. [EKKP18] is a collection of selected papers highlighting different fields in which solving optimization problems play an important role including disciplines like data analysis, game theory, and economic applications. Therefore, the importance of solving optimization problems is undeniable.

The goal of solving an optimization problem is to compute the optimal parameter combinations for a possibly complex system such that a given cost function is optimized. In our work, the cost function is given by an analytically known objective function which we aim to minimize or maximize respectively. In real-world applications, the algebraic representation of the objective function is usually unknown. Thus, one can only evaluate the objective function for specific inputs to get the corresponding function value. Usually, this type of optimization is called black-box optimization because the objective function is not given and therefore is a black-box itself. Under these circumstances, the classical optimization approaches, e.g. using first- or higher-order derivatives to solve the optimization task are very expensive or even unfeasible.

Another method for solving black-box optimization problems without using derivatives is random search. One chooses a large set of random input parameter combinations x and calls the function f with those inputs to get the corresponding set of outputs. Thereafter, one would iterate over the set of outputs and pick the largest or smallest among these values. Typically, one must assume that evaluating a black-box function might be very costly and time intensive due to its unknown nature. According to this, random search is computationally very complex and especially in higher-dimensional setups, it is not feasible due to the curse of dimensionality. There are many more zero-order optimization algorithms like random search that do not rely on derivatives, however, these algorithms are not always well suited for specific tasks they were not designed for. This makes it necessary to pick a more sophisticated search approach. The manual development of sophisticated optimization algorithms is a challenging and tedious process.

This makes the learning-to-optimize approach appealing because it allows us to automatically learn a task-specific optimization strategy that can handle higher-dimensional setups. For example, the authors of the papers [LM16], [SC16] and [ADG+16] automate the development of an optimization algorithm by using reinforcement learning. In this matter, they describe an optimization problem as a Markov decision process and the learned optimization algorithm is represented by a policy. We use a similar approach in our work.

To avoid ambiguity we will refer to the resulting learned optimization algorithm as the "learned algorithm" or "policy". We will refer to the reinforcement learning algorithm that learns a policy as the "learner" or "(reinforcement) learning algorithm".

The scientific works [LM16], [SC16] and [ADG+16] show that learned algorithms can outperform hand-crafted optimization algorithms in terms of convergence speed and final objective function value.

However, state-of-the-art optimization algorithms have limitations concerning constraints on input parameters. With current optimization algorithms, we would have to perform the entire reinforcement learning process again for every possible constraint combination on input parameters. This is infeasible, because of the enormous memory and time consumption this would imply (see [DPRL21]).

In this work, we explore the possibility of introducing additional constraints on a specific set of input parameters we call “free parameters”. The constraints define possible values of “free parameters”. We refer to all other inputs without constraints as “optimization parameters”. Essentially, we want to learn the function that maps possible values of “free parameters” to the corresponding optimal input which in turn maximizes or minimizes the objective function with respect to the values of our “free parameters”. Moreover, we analyze the scalability to high-dimensional setups to explore the limitations of such approaches.

We give an explanation for the concept of “free” parameters with a real-world example in post-silicon validation:
Suppose a semiconductor has three operating modes. We might be interested in the optimal operating mode, given a specific temperature in a room. Here, the parameter representing the temperature is part of the set of “free parameters” and the parameter representing the operating mode is part of the “optimization parameters”. Running our reinforcement learning algorithm results in outputting an optimization algorithm telling us which of the three operating modes is optimal for a given temperature. If we take this semiconductor to another room with a different temperature we would not want to be forced to run our reinforcement learning algorithm again but rather that our learned optimization algorithm optimizes the operation mode depending on the temperature.

1.1 Research Contributions

We implemented a reinforcement learning approach to learn optimization algorithms for black-box objective functions. Further, we implemented evaluation metrics and visualization methods to analyze the performance of our policies. Additionally, we selected and implemented common objective functions for testing optimization algorithms and their gradients. For our purpose, only objective functions which are scalable in dimension are suitable. We evaluated and compared the performance of our learned algorithms with different sets of optimization and free parameters. Subsequently, we analyzed the behavior of our learned algorithms in a high-dimensional setup.

1.2 Outline

In the following Chapter 2, we present related work that has been conducted on the topic of learn-to-optimize and data-driven algorithm design. Chapter 3 builds the foundation for this work and introduces important concepts, terms, and technologies. We specify our problem formulation, present our study design and include implementation details in Chapter 4. In Chapter 5 we describe the experiments we conducted including the results we obtained. Further, we interpret and discuss them in this chapter. Afterward, we point out the limitations of this work and propose suggestions for improvement and follow-up work in Chapter 6. This thesis ends with the conclusion in Chapter 7.

2 Related Work

The key idea of related research areas is to formulate the design of an optimization algorithm as a learning problem. Most commonly, reinforcement learning is used in the learning process. There are three main goals of this data-driven approach called learn-to-optimize. The first goal is to improve the performance of existing hand-crafted optimization algorithms like Adagrad [DHS11] and its variations Adadelta [Zei12], Adam [KB14] and Nadam [Doz16]. Second, learn-to-optimize makes it possible to adapt existing algorithms to specific optimization problems. Finally, learn-to-optimize automatizes the design of optimization algorithms so that no human expertise on the topic of optimization algorithms is needed to develop an optimization algorithm.

In the article [LM16], Li and Malik introduced the framework for describing the design of algorithms as a learning task. The authors claim that no one before tried to automatically learn algorithms that outperform existing optimization algorithms. In their work, they outline that every optimization algorithm follows a similar structure. The crucial part differentiating two optimization algorithms is the update formula which is used to determine the next parameter values in the domain to explore. They model this update formula using a neural network (NN) as powerful function approximators that work well in high-dimensional settings. NNs are parameterized models that can describe arbitrary (non-linear) functions. Learning the update formula is done by adjusting the weights of the neural network using reinforcement learning algorithms such as guided policy search. A desirable or good optimizer converges as fast as possible and as accurately as possible to the desired minimum or maximum. They use the sum of all obtained objective function values as feedback to reward good optimizers and penalize bad ones. The higher this metric, the better the performance of the optimizer, assuming the task is a maximization task. They demonstrate that their resulting algorithm outperforms existing hand-developed algorithms for several classes of both convex and non-convex objective functions when it comes to convergence speed and final objective function value. Furthermore, Li and Malik state that learned algorithms are generally preferable to hand-developed optimization algorithms since learned algorithms make no assumptions about the underlying objective function beforehand. Moreover, automatically learned optimization algorithms do not require hyperparameter-tuning after training.

In the paper [ADG+16] Andrychowicz et al. cast the design of optimization algorithms as a learning problem following the approach in the article [LM16]. Long short-term memory neural nets (LSTMs) are a special form of recurrent neural networks (RNNs). They were first introduced in the scientific work [HS97]. The idea of LSTMs is to add the dimension of time and memory structure to an RNN. The authors decided to implement the learned algorithms using LSTMs and found that the resulting learned algorithms achieve better results than generic hand-developed alternatives. They show performance improvements for the objective function on which the learned algorithms were trained, but also for other objective functions which have similar structures to the original objective function.

In the study [SC16] the authors apply the learning-to-learn approach on optimizing black-box functions. The field learning-to-learn can be seen as a special case of learning-to-optimize. One speaks of learning-to-learn when learning-to-optimize is used on objective functions that describe loss functions of other learning algorithms. The authors compare the learned optimization algorithms to Bayesian optimization (BO) and focus on the duration of computation, the ability to generalize to longer horizons, and the trade-off between exploration and exploitation. In addition to LSTMs, they also utilize another type of RNN in their implementation, namely differentiable neural computers (DNCs). These networks were first introduced by Graves et al. in the scientific work [GWR+16]. Their results show that up until a certain training horizon the resulting algorithms perform at least as good as Spearmint, a commonly used BO package. Furthermore, they demonstrate that RNNs are capable of producing results faster than BO. However, they point out that learned optimization algorithms have disadvantages as well. In particular, they mention that learned optimizers do not perform as well as BO when evaluating for training horizons that are longer compared to the training horizon used at training time. The training horizon refers to the number of steps in one episode. The same issue was covered by Duan et al. in the work [DSC+16].

In the paper [DPRL21], Domanski, Pflüger, Rivoir, and Latty managed to efficiently and robustly solve complex tuning tasks in post-silicon validation using learn-to-optimize. In their last chapter, the authors mention a difficulty they encountered while conducting their work. They noticed that it becomes infeasible with state-of-the-art optimization algorithms to optimize parameters of functions that depend on known conditions. We mentioned an example in 1 to illustrate what such a function could look like. Hand-crafted algorithms are very slow and lead to unreasonably high runtime and large memory consumption in many cases. The reason for this is the need to re-run the optimization algorithm for every possible combination of values for the conditions. They motivate that it would be of interest to tackle this difficulty.

This bachelor thesis aims to investigate and solve this issue. Another focus of this work is the scalability of the proposed approach to higher dimensional settings and an increasing number of conditions.

3 Fundamentals

This chapter builds the foundation of this work and introduces important concepts, terms, and technologies.

3.1 Optimization

Mathematical optimization and solving optimization problems involves methodically selecting arguments with the most preferable values out of a specified domain. This general notion takes a wide range of diverse input domains and objective functions into account. In our setting, we aim to minimize an objective function.

The optimal input x^* is the argument x out of a domain X which minimizes a function f.

$$x^* = \arg\min_{x \in X} f(x) = \{x \in X \mid f(x) = \min_{x' \in X} f(x')\} \tag{3.1}$$

Scientists mainly use three approaches to solving optimization problems. First, they can employ heuristics, which offer approximations for the solution. Second, there are algorithms which finish after a certain amount of steps. Third, iterative approaches that converge to a solution are able to approach closer to the optimal solution with more iteration steps. In this work, we will use reinforcement learning (an iterative approach) to solve optimization problems.

3.1.1 Black-Box Optimization

A black-box is a model that only reveals its in- and outputs. Black-boxes find application in many scientific fields including optimization. The inner structure of the black-box is not transparent (metaphorically black). Black-box functions f_b describe objective functions that are analytically not known to the entity interacting with f_b.

Black-box optimization is a special case of optimization problems. This field of study deals with optimizing black-box functions. Standard approaches for black-box optimization include feeding the black-box with inputs and observing its outputs. The observed behavior is used in order to assess the inner functionality of the black-box.

3.2 Artificial Intelligence

Artificial intelligence (AI) tries to mimic the natural intelligence of living beings with the help of machines. [Lug05] AI research mainly focuses on studying intelligent agents. Intelligent agents are systems which understand their surroundings and act in a way that maximizes the chances of succeeding in their goals.

3.2.1 Machine Learning

Machine learning (ML) is a core part of AI. [MBD+90] ML is an area of research focused on comprehending and developing "learning" processes which use data to enhance performance on a given task. Programs which use ML can do tasks without having them explicitly coded. For easy tasks, it is feasible to build algorithms which instruct the device how to carry out all the steps necessary to address the issue at hand. No learning is required on the side of the computer. However, it might be difficult for humans to develop algorithms for increasingly complex tasks manually. In reality, it may be more beneficial to assist the computer in creating its own algorithm than to have human programmers define each necessary step.

The ability of learners to draw generalizations from their experience is a key goal in ML. In this application, generalization refers to a learning machine's capacity to perform well on tasks which the learning algorithm has never seen before. In order for learners to make sufficiently precise predictions in novel situations, they have to develop a generic model of the space of possible problem instances.

The premise behind learning algorithms is that approaches, methods, and conclusions which have proven successful previously are likely to do so again in the future.

The discipline of ML benefits from the tools, theory, and application fields which come from the field of mathematical optimization. Many learning tasks are expressed as the minimizing of a particular loss function. The loss function describes the gap between the actual circumstances and the model's predictions.

Reinforcement Learning

Reinforcement learning (RL) is a subfield of ML. [SB18] In RL, one assumes that there is an agent which is located in an environment. At each step, the agent performs an action and receives a state and a reward from the environment. An observation is the part of the state that can be observed by the agent. The agent chooses the action with the help of a so-called policy. The goal is that the agent learns a promising policy through this process. Figure 3.1 shows a visual representation of this interaction loop. In each so-called episode, the agent's initial state is drawn randomly from a distribution. The interaction cycle continues until the environment reaches a final state. Then the next episode begins. The reached cumulative reward per episode of an agent is called return. The goal of an RL algorithm is to maximize the return over all time steps as the agent interacts with the environment.

A reinforcement learning problem is typically formally represented as an Markov decision process (MDP).

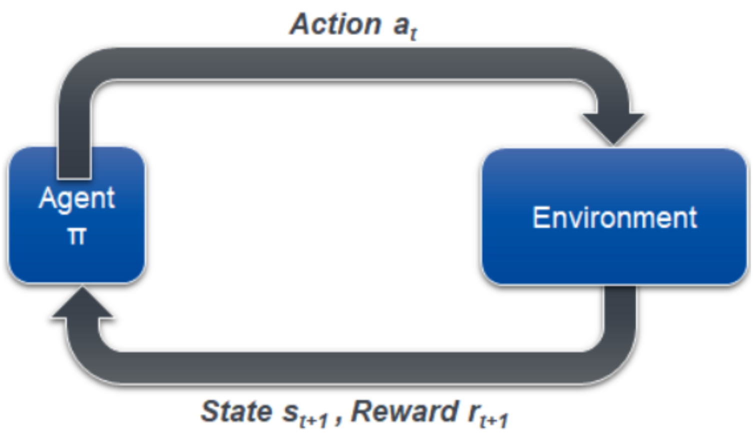


Figure 3.1: Reinforcement learning loop containing agent (left) and environment (right) interacting

Markov Decision Process

Markov Decision Processes (MDPs) offer a foundation for modeling situations where decisions have to be made in circumstances where the results are partially determined by chance and partially controlled by the entity making the decisions. In the case of RL, the agent is the entity making decisions. They find application in the research of optimization problems. MDPs were already known before 1960. [Bel57]

We consider a finite-horizon MDP with continuous state and action spaces defined by the tuple $(S, A, p_0, p, c, \gamma)$, where

- S is the set of states,
- A is the set of actions,
- $p_0 : S \longrightarrow R^+$ is the probability density over initial states,
- $p : S \times A \times S \longrightarrow R^+$ is the transition probability density, that is, the conditional probability density over successor states given the current state and action,
- $c : S \longrightarrow R$ is a function that maps state to cost and
- $\gamma \in (0, 1]$ is the discount factor

The objective is to learn a stochastic policy $\pi^* : S \times A \longrightarrow R^+$, which is a conditional probability density over actions given the current state, such that the expected cumulative cost is minimized

First-order methods use a π that depends only on the gradient of the objective function, whereas second-order methods use a π that depends on both the gradient and the Hessian of the objective function.

The MDP is always in a certain state s. Each state defines which actions the agent may choose. After choosing an action a, the MDP enters a new state s', according to a, and the agent receives a reward $R_a(s, s')$ corresponding to s'.

The selected action a has an impact on the likelihood that the MDP enters a certain state s'. It is provided by a state transition function $P_a(s, s')$. The action choice of the agent a and the present state s determine the subsequent state s'. State transitions of MDPs are independent of prior actions and states. Only a and s are relevant for determining s'. This is called the Markov property. [OW12]

3.3 Neural Network

An artificial neural network (NN) is a model built on a set of interconnected nodes called "artificial neurons" which imitate neurons in a human brain. [Pic94] However, there is one big difference between NNs and human brains. The number of nerons and their connections are static in NNs. Human brains have more dynamic structures. NNs form directed, weighted graphs as can be seen in Figure 3.2.

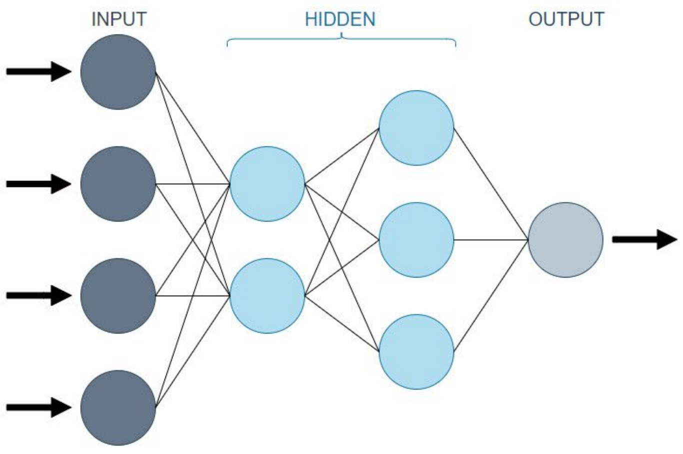


Figure 3.2: Example of an NN with four input nodes, two hidden layers of different sizes, and one output node

A NN consists of multiple connected neurons. Similar to the synapses in human brains, each connection allows information to travel between neurons. This information is called "signal" and it is a continuous real number. After processing a signal, an artificial neuron can signal other artificial neurons which are linked to it. We call the connections between artificial neurons "edges". Edges contain weights that change as learning progresses. Weights are multiplicative factors which increase or decrease a connection's signal intensity. A neuron N receives weighted input from other neurons and can transform it in a non-linear manner. The weights are adjusted gradually by an algorithm operating on training data. This is the process often referred to as "learning". Normally, artificial neurons are grouped into layers. Distinct layers may modify their inputs in a variety of ways. Signals go through the layers, perhaps more than once, from the input layer to the output layer. NNs can model almost any arbitrary continuous function and recognize patterns in the data it is working on. Typically, NNs "learn" to execute tasks by operating on examples, without having any rules, specific to a task, written into them.

3.3.1 Recurrent Neural Network

Recurrent Neural Networks are a special type of NNs. The output from a certain node N can influence the future input of the same node N in RNNs. It is possible that a set of edges builds a cycle. RNNs have the capacity to handle inputs of varying lengths by using the internal memory. Figure 3.3 shows an illustration of a compressed RNN and its unfolding.

RNNs can be extended via further stored states. These stored states, which are directly governed by the RNN, are called gated states. They are one component of long short-term memory networks (LSTMs). The signal will pass through a layer of a recurrent neural network (RNN) multiple times. When RNNs are trained by a method using gradients (like gradient descent), long-term back-propagated gradients can go to zero or diverge to infinity. This is called the vanishing gradient problem. [PMB13] Most of the time, using long short-term memories prevents this problem.

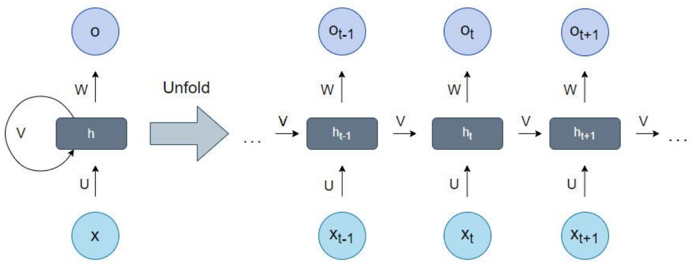


Figure 3.3: From left to right: Compressed and unfolded versions of a simple RNN

Long Short-Term Memory

Long short-term memory is an extension of RNNs used in deep learning and AI. LSTM features so-called "feedback connections" as opposed to typical feedforward NNs. RNNs including LSTMs may analyze complete data sequences in addition to individual data points.

RNNs in their standard version have "short-term memory" as well as "long-term memory". Similar to how synaptic adjustments in the body preserve long-term memory, the network's weights change once throughout each training episode. "Long short-term memory" refers to the short-term memory that LSTMs seek to add to RNNs which can endure several hundred time steps.

The main components of LSTMs are the cell, as well as an input-, an output- and a forget gate. This can be seen in Figure 3.4. The gates control how information enters and leaves the cell, and the cell is responsible for remembering values for an indefinite time.

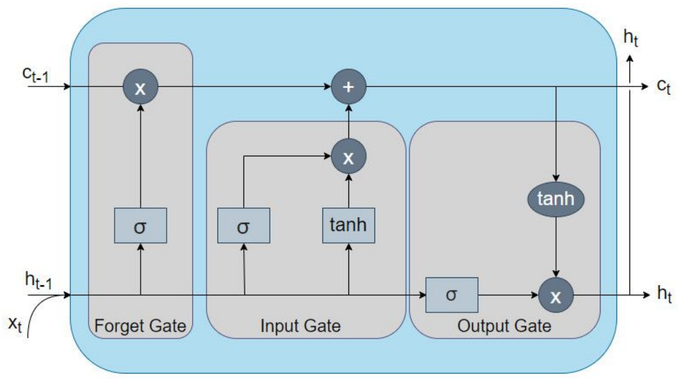


Figure 3.4: Body of the internal structure of an LSTM architecture including input-, output- and forget-gate

The purpose of LSTMs is to solve the vanishing gradient problem which might arise when training conventional RNNs. Theoretically, conventional RNNs are able to keep track of input dependencies that can be arbitrarily long. There is one issue with traditional RNNs in practice. While training a traditional RNN via back-propagation, gradients that are back-propagated have the potential to converge to zero ("vanish") or diverge to infinity ("explode") in the long-term. This happens because computations operate on numbers with finite precision. Due to the ability of LSTMs to propagate gradients without changing them, RNNs utilizing LSTMs can mostly overcome the vanishing gradient problem. Errors may instead go backward over a limitless number of "layers" that are unfolded in space. [HS96] [GSC00]

4 Methodology

In this chapter, we specify our problem formulation and present our study design. We define the objective functions we chose and explain the high-level structure of our implementation as well as some details. After, we will introduce the external libraries we utilized. Lastly, we highlight some methodological limitations of this work and how we addressed them.

4.1 Problem Formulation

In this section, we elaborate on how we cast an optimization problem as a reinforcement learning problem. The pseudocode for an optimization algorithm is displayed in Algorithm 4.1. Our learned optimization algorithms start by picking a random point x_0 out of the domain. This point has to fulfill the set of constraints C. After, the point is updated iteratively using the update function π until either some stopping condition is met or the end of the for loop is reached. The point that was chosen last is outputted. The update formula π is modeled by the policy of the agent in reinforcement learning.

Algorithm 4.1 Optimization Algorithm Pseudocode

Input: Number of iterations, domain, input dimension, objective function, set of constraints on ´´free" parameters
Output: Minimum of the objective function

```
 1: $N_{\text{episode}}$ := number of iterations
 2: $d$ := input dimension
 3: $D$ := domain
 4: $C$ := set of constraints on ´´free" parameters
 5: $f$ := objective function
 6: $x_0$ := random value from D respecting C
 7: for i in $1, \dots, N_{\text{episode}}$ do
 8:     if stopping condition is true then
 9:         return $x_{i-1}$
10:     end if
11:     $x_i := \pi(C, f, \{x_0, \dots, x_{i-1}\})$
12: end for
end
```

We introduce the term "slice-possibilities" as we will need this thought later. We are interested in objective functions that have "free" parameters as input. "Free" parameters are input parameters that are fixed through constraints. For example, suppose we want to optimize the two-dimensional Sphere function $f(x_1, x_2) = x_1^2 + x_2^2$ and x_1 is a ´´free" parameter. Let $a \neq b$ be two random but known values. For the constraint $x_1 = a$ we would then have to optimize the function $f(x_1, x_2) = f(a, x_2) = a^2 + x_2^2 =: g(x_2)$. This resulting function g has only one optimization parameter left. The plane p that satisfies the constraint $x_1 = a$ and is parallel to the $f(x_1, x_2) - x_2$-plane includes all points that are possible when respecting the constraint. The function g also occurs when we would intersect the original objective function f with the plane p. The result of this intersection would be a "slice" of the original objective function. Therefore, one can think of a specific constraint on a "free" parameter as specifying a slice of the objective function. A different constraint (for example $x_1 = b$) would result in a different "slice". The number of "slice-possibilities" refers to the number of possible "slices". If in the same example the constraint would be $x_1 \in \{a, b\}$, there would be two "slice-possibilities". This example is extendable to multiple input dimensions and multiple "free" parameters. It is worth mentioning that the number of "slice-possibilities" grows rapidly for higher numbers of "free" parameters due to the resulting combinatorial possibilities. In the case of input dimension ten with nine "free" parameters, where every "free" parameter has 20 possible values, there would be 20^9 "slice-possibilities".

Any kind of learning involves practicing with a limited number of instances out of a larger set of examples. After the training is done, the learned algorithm tries to apply what has been learned to unseen instances. In our setting, the large set contains all potential "slice-possibilities". During training, we pick random instances out of this set. For evaluation, we pick values out of a grid. Different instances of "slice-possibilities" correspond to different constraints on our "free" parameters. Hence, our learned optimization algorithms can optimize various possible combinations of constraints on "free" parameters. We refer to this as the ability of learned algorithms to generalize.

We use different methods to select "slice-possibilities" at training-time and evaluation-time because otherwise the learned optimizers would memorize the minima for the selected "slice-possibility" once found. Such an optimization algorithm achieves to output the minimum in a single step for every potential selection. Selecting multiple combinations out of all "slice-possibilities" would not prevent this. In this case, the learned optimization algorithm would be able to tell which "slice-possibility" was chosen after an episode-steps and output the memorized minimum directly after. This behavior is not desirable because of the following reason. It is more time-consuming to learn optimization algorithms than it is to run a conventional optimizer for a given "slice-possibility" due to the fact that memorizing minima entails locating them beforehand. Consequently, the learned optimization algorithm needs the ability to generalize on unseen "slice-possibilities" which are distinct from ones encountered in the training procedure. Learned optimization algorithms have a big advantage over non-learned optimization algorithms due to generalization. [Li17]

Our approach is intended for black-box optimization as described in Section 3.1. However, we exclusively considered analytically known objective functions. This gives us the ability to verify the approach more easily, especially with respect to scalability in higher dimensional settings.

4.2 Study Design

We define all relevant aspects, terms and concepts including the objective functions we used, the components of our program structure, and external libraries we utilized in our experiments in this section. Our code is written in the programming language Python and we are using version 3.8.

4.2.1 Program Structure

The high-level program structure of our program is illustrated in Figure 4.1. First, we set all

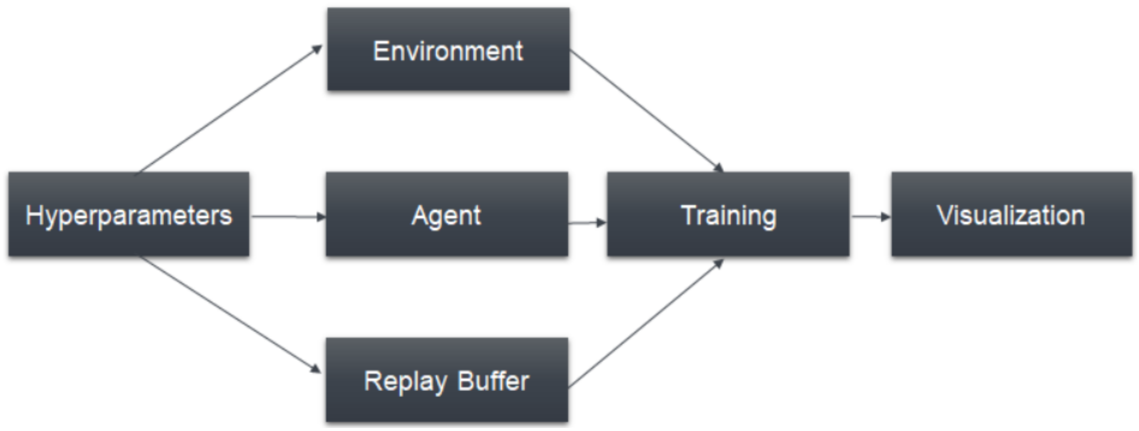


Figure 4.1: Overview of our program structure

mandatory hyperparameters. Then we initialize the environment, the agent, and a replay buffer, i.e. all components that are necessary for the subsequent training. After that, the actual training is executed. In the final step, we visualize the results. In the following, we will discuss the hyperparameters, the environment, the agent, the training, and the evaluation in greater detail.

4.2.2 Hyperparameters

Hyperparameters are parameters that control and steer the learning process of an optimization algorithm. They are already fixed before the training begins and do not change during training as opposed to parameters like the weights of the neural network. In addition, hyperparameters have no impact on the performance of the resulting optimization algorithm. They exclusively influence properties of the learning process such as the speed or quality of learning an optimization algorithm.

Listed in Table 4.1 are the most relevant parameters and hyperparameters and their default values. All listed parameters and hyperparameters have to be natural numbers and the number of “free” parameters N_{free} cannot exceed the input dimension d.

Parameters and Hyperparameters		
Name	Symbol	Domain
Training iterations	$N_{\text{iteration}}$	10000
Input dimensions	d	2
Number “free” parameters	N_{free}	1
Episode length	N_{episode}	50
Observation horizon	$N_{\text{observation}}$	1
Batch size	N_{batch}	512

Table 4.1: The most important parameters and hyperparameters alongside their abbreviation symbol and their domain

4.2.3 Objective Functions

In this subsection, we list all of the objective functions we used to train our optimization algorithm in alphabetic order. All of these are common and well-known functions that are often used for testing optimization algorithms. For functions that are configurable via parameters, we used recommended standard values found in the literature. [MS05] For every objective function, we further provide the associated gradient in Appendix A.

Ackley Function

The Ackley function is almost flat in its outer region and its global minimum lies in a large central hole. The outer region contains multiple evenly distributed local minima, in which optimization algorithms might get trapped in. We used $a = 20, b = 0.2$ and $c = 2\pi$

$$f(x) = -a \cdot \exp\left(-b \cdot \sqrt{\frac{1}{d} \cdot \sum_{i=1}^{d} x_i^2}\right) - \exp\left(\frac{1}{d} \cdot \sum_{i=1}^{d} \cos(c \cdot x_i)\right) + a + \exp(1) \tag{4.1}$$

Griewank Function

The Griewank function owns a lot of local minima, spread over the entire domain. The surface area of the Griewank function has a very fine texture.

$$f(x) = \sum_{i=1}^{d} \frac{x_i^2}{4000} - \prod_{i=1}^{d} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1 \tag{4.2}$$

Levy Function

The Levy function has the most complex gradient of our functions. Let $\forall i \in \{1,\dots,d\} : w_i = 1 + \frac{x_i - 1}{4}$. Then the function and its gradient can be defined as:

$$(4.3)\quad f = \sin^2(\pi \cdot w_1) + \sum_{i=1}^{d-1}\Big((w_i - 1)^2 \cdot \left[1 + 10\sin^2(\pi \cdot w_i + 1)\right]\Big) + (w_d - 1)^2 \cdot \left[1 + \sin^2(2\pi \cdot w_d)\right]$$

Rastrigin Function

The Rastrigin function owns multiple regularly distributed local minima.

$$(4.4)\quad f(x) = 10 \cdot d + \sum_{i=1}^{d}\left[x_i^2 - 10 \cdot \cos(2\pi \cdot x_i)\right]$$

Rosenbrock Function

The Rosenbrock function is a member of the valley-shaped functions. It is also called Rosenbrock's valley or Rosenbrock's banana function. The global minimum lies in a narrow, oblong, parabolic-shaped valley. Optimization Algorithms tend to find this valley rather quickly, but struggle to find the global minimum inside. [PWG13]

$$(4.5)\quad f(x) = \sum_{i=1}^{d-1}\left[100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2\right]$$

Sphere Function

The Sphere function is the simplest among our objective functions. This bowl-shaped function has no minimum except for the one global minimum in the center of its domain. Moreover, this is the only strictly convex function we used.

$$(4.6)\quad f(x) = \sum_{i=1}^{d} x_i^2$$

Styblinski-Tang Function

The Styblinski-Tang function has a unique curvy bowl shape.

$$(4.7)\quad f(x) = \frac{1}{2} \cdot \sum_{i=1}^{d}(x_i^4 - 16x_i^2 + 5x_i)$$

Zakharov Function

The Zakharov function is the only representative of plate-shaped functions in our collection. It has no local minimum besides the global one.

$$f(x) = \sum_{i=1}^{d} x_i^2 + \left(\frac{1}{2} \cdot \sum_{i=1}^{d} ix_i\right)^2 \left(\frac{1}{2} \cdot \sum_{i=1}^{d} ix_i\right)^4 \tag{4.8}$$

Of the many available functions we could have chosen, we picked the ones we did for two main reasons. First and most importantly, all of the shown objective functions are dimensionable, i.e. they are easily adaptable to higher dimensions by incrementing the variable d. This is essential because this gives us the ability to conduct experiments in high-dimensional setups to test the limits to which our proposed approach yields acceptable results.

Second, we made sure that these particular objective functions have as little in common as possible concerning similarities in their shapes and properties besides dimensionability. We did this to verify that our proposed approach is well-suited for different kinds of objective functions.

We evaluated all objective functions on their typical search domain, which we summarized alongside the global minimum and the corresponding x* in Table 4.2. For example, the Ackley function is usually evaluated on the hypercube $\forall i \in \{1, \dots, d\} : x_i \in [-32, 32]$ and has its global minimum at $f(\text{x*}) = 0$ with $\text{x*} = (0, \dots, 0)$. We are not able to specify a universal codomain for the objective functions as the codomain is dependent on the input dimension d.

In Figure 4.2 we provide a graphical illustration of the eight objective functions we used for two input dimensions.

Objective functions			
Function	Search domain	Global minimum $f(x^*)$	Corresponding input x^*
Ackley	$-20 \le x_i \le 20$	0	$(0, \dots, 0)$
Griewank	$-600 \le x_i \le 600$	0	$(0, \dots, 0)$
Levy	$-10 \le x_i \le 10$	0	$(1, \dots, 1)$
Rastrigin	$-5 \le x_i \le 5$	0	$(0, \dots, 0)$
Rosenbrock	$-2 \le x_i \le 2$	0	$(1, \dots, 1)$
Sphere	$-5 \le x_i \le 5$	0	$(0, \dots, 0)$
Styblinski-Tang	$-5 \le x_i \le 5$	$-39.2 \cdot d$	$(-2.9, \dots, -2.9)$
Zakharov	$-5 \le x_i \le 5$	0	$(0, \dots, 0)$

Table 4.2: Used objective functions alongside their search domain, global minimum, and the corresponding input

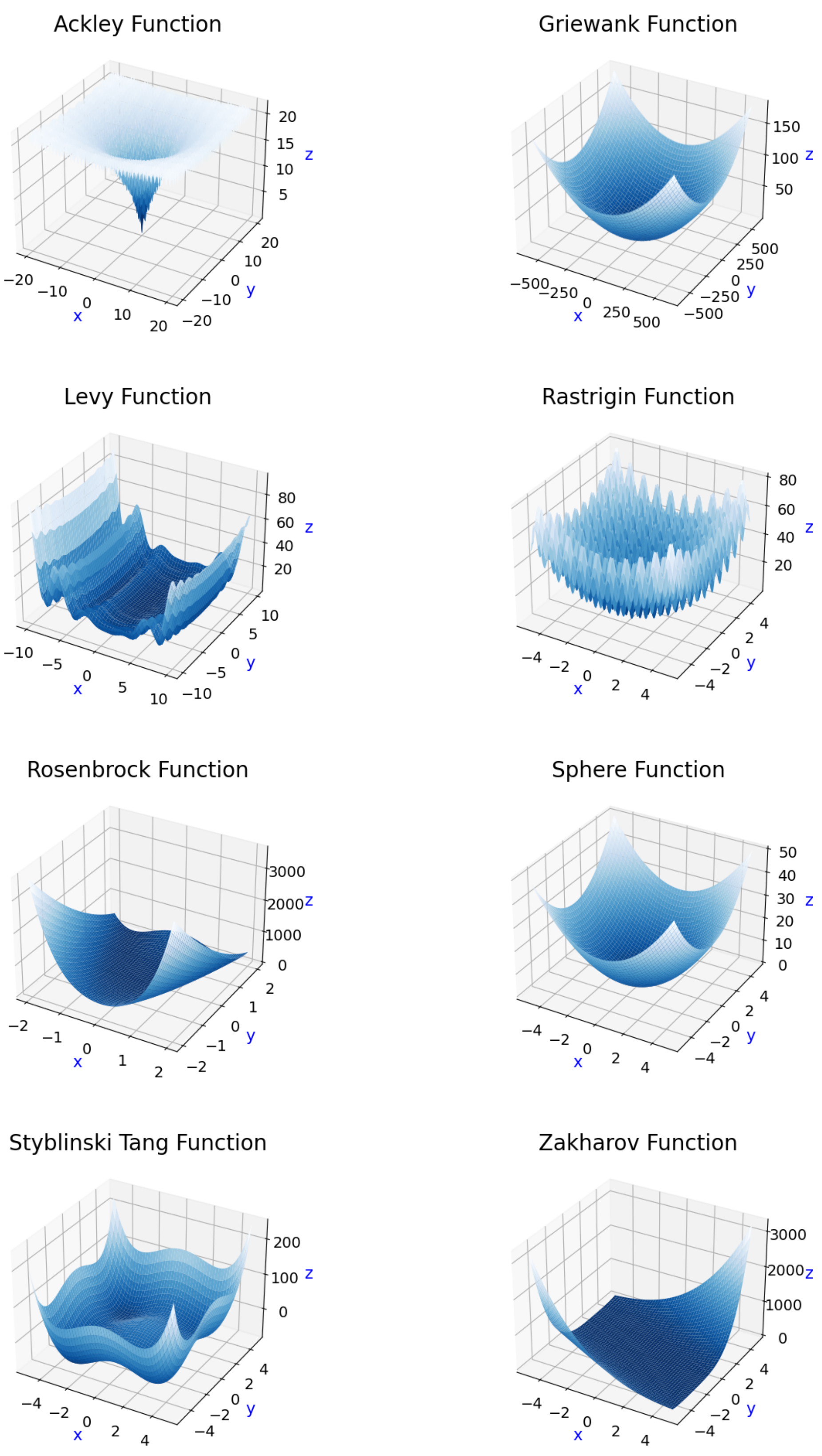


Figure 4.2: Graphical illustration of the eight objective functions for two input parameters

4.2.4 Environment

In this subsection, we discuss the main components of the reinforcement learning environment. We define valid actions of the agent and the structure of the observations that the agent gets from the environment. The two most significant methods characterizing an environment are the reset- and the step-method.

Action

Actions are interpreted as input parameters for objective functions. In our setting, some of the input parameters are "free" parameters that are fixed by constraints. The agent has to respect these constraints when choosing an action. Therefore, we define an action a to represent a $N_{\text{optimization}}$-dimensional vector in a continuous space, where $N_{\text{optimization}} = d - N_{\text{free}}$. Thus, an action consists of $N_{\text{optimization}}$ representing values for input parameters that are not "free".

We specify valid values for an action by setting lower and upper boundaries for every component of an action. All values in between are possible. Every objective function has a different domain on which it is usually evaluated. Despite that, we choose a uniform action-space $\forall i \in \{1, \dots, N_{\text{optimization}}\} : -1 \leq a_i \leq 1$. This way, we do not have to modify the action-space for every time we want to observe a different objective function. Instead, when evaluating the objective function, it gets an action a as input and scales it to its native domain of the objective function internally.

Let us assume that the variables lower and upper are the boundaries for an objective function. We map the value $a_i \in [-1, 1]$ to the interval [lower, upper] with the following formula:

$$a_{i_{\text{new}}} = (a_i - (-1)) \cdot \frac{\text{upper} - \text{lower}}{1 - (-1)} + \text{lower} = (a_i + 1) \cdot \frac{\text{upper} - \text{lower}}{2} + \text{lower} \tag{4.9}$$

As a result, the agent has the option to evaluate every point on a given slice of the objective function at anytime.

Observation

In our setting, an observation is composed of $N_{\text{observation}}$ tuples, each consisting of a position in the domain x and the corresponding normalized reward r. A position in the input domain consists of d values and can be split into two parts; the "free" parameters and the optimization parameters. The first N_{free} values are determined by the constraints on the "free" parameters. The remaining $N_{\text{optimization}}$ parameter values are determined by the chosen action of the agent. In total, one tuple consists of $N_{\text{free}} + N_{\text{optimization}} + 1 = d + 1$ values. The Observation that the agent receives from the environment after step i has the following form:

$$\begin{aligned} O_i &= \left\{(x_{i-N_{\text{observation}}+1}, r_{i-N_{\text{observation}}+1}), \dots, (x_i, r_i)\right\} \in \mathbb{R}^{(d+1)\cdot N_{\text{observation}}}, \text{ with} \\ &\forall i < N_{\text{observation}} : x_{i-N_{\text{observation}}+1} = (0, \dots, 0) \text{ and } r_{i-N_{\text{observation}}+1} = 0. \end{aligned} \tag{4.10}$$

Hence, the agent is constantly aware of its previous $N_{\text{observation}}$ domain positions and the corresponding rewards.

Reset-method

Before an environment is ready for usage, the reset-method must be called. The reset-method is called to reset an environment into an initial state. In this initial state, no steps have been taken. The agent starts at a random location on the objective function picked from a uniform distribution. All obtained observations in previous episodes are deleted and an initial observation is built. This method gets no inputs and returns an initial observation to the calling instance.

Step-method

The step-method is called to execute one action in an environment. This incorporates updating all variables properly as well as returning the response of the environment. This method gets an action as input and returns a triple containing the associated updated observation, the associated reward, and a done-flag.

At every call of the step-method, we count how many times this has already happened. Further, we update the current observation. Lastly, we check if the current episode terminated. We declare an episode to be finished when the step-method has been called more than N_{episode} times. That means an episode is always exactly N_{episode} steps long. After, the next episode begins.

4.2.5 Agent

Every reinforcement learning algorithm learns by letting the agent interact with the environment. The agent collects experience in the form of episodes through this process. As explained in Subsubsection 3.2.1, reinforcement learning agents use policies to select their next action. Agents are mainly distinguished between those who pursue an on-policy strategy and those who pursue an off-policy strategy.

On-policy algorithms throw away the data collected so far after each episode ends. On the other hand, off-policy algorithms use data collected over several episodes. This means it is not thrown away after each episode. This leads to the fact that older data is kept for learning. The REINFORCE-algorithm [Wil92] and the Proximal Policy Optimization-algorithm [SWD+17] are two well-known on-policy algorithms. The most well-known off-policy algorithms are the Deep Q Network- [MKS+13], Soft Actor-Critic- [HZAL18], Deep Deterministic Policy Gradient- [LHP+15] and the Twin Delayed Deep Deterministic Policy Gradient-agent [FHM18].

In general, on-policy algorithms promise better convergence but worse sample efficiency, whereas off-policy algorithms tend to yield worse convergence and better sample efficiency. Worse convergence means that the algorithm does not find the optimal solution at all or only very slowly. The term sample efficiency describes how much data has to be collected to learn. Good sample efficiency means that little data is needed for learning or that the algorithm extracts much information out of little amounts of data.

In this work, we limit ourselves to on-policy algorithms. We conducted experiments with the REINFORCE- and the PPO-agent. The library TF-Agents offers off-the-shelf implementations for these.

4.2.6 Training

One training iteration step consists of the following steps: Initially, we gather multiple trajectories using the collect-policy of the agent. The difference between the collect-policy and the "normal" policy of the agent is that the collect-policy adds randomization to the action selection to encourage environmental exploration. The replay buffer is the data structure in which the accumulated experiences of the agent in the form of trajectories can be stored. The agent gets the data from the replay buffer and completes one training step once enough trajectories have been gathered. One training step of the agent updates the policy and the underlying internal weights of the NN. The next step is the clearing of the replay buffer. This last step is characteristic for on-policy algorithms like REINFORCE and PPO. After, the next training iteration step begins. The number of training iteration steps is determined by the parameter $N_{\text{iteration}}$.

4.2.7 Evaluation

For the evaluation of our learned optimization algorithms, we mainly inspect two metrics. We analyze the normalized training rewards and the mean squared error (MSE) over the number of training steps. We inspect the MSE and not only the normalized reward because we might have defined the normalized training reward differently. In contrast to information derived by interpreting the normalized training reward, the MSE shows the ground truth. The MSE shows actual proven information about the distance of the optimization algorithms prediction and the actual minimum on a "slice" through empirical evidence.

When evaluating, we do not want the performance of an optimization algorithm to be dependent on the random start point it gets. For this reason, we build a grid of starting points covering the entire input domain. We evaluate the resulting optimization algorithms for every starting point and build the average value. To keep the computational complexity reasonably low, we chose to pick four values per dimension. In total this results in $N_{\text{start}} = 4^d$ starting points. This means, we cover $4^{N_{\text{free}}}$ "slice-possibilities" for N_{free} "free" parameters.

After the training for a specific objective function, input dimension, and number of "free" parameters is done, the most recent policy is used for evaluation. This policy outputs a final objective function value after N_{episode} steps. In the following, we will refer to this value as the predicted minimum.

Reward

We want to reward optimization algorithms that converge quickly to the minimum on a given slice of an objective function f. We chose

$$(4.11)\quad r_i = \frac{\max - f(x_i)}{\max - \min} \in [0, 1]$$

as a normalized training reward where max and min are the extreme values on the given slice and x is the proposed location by the learned optimization algorithm.

Reinforcement learning algorithms try to maximize the reward. By maximizing r_i, the actual objective function value $f(x_i)$ is minimized for a given slice as a fixed slice fixes the values max and min. This means low objective function values lead to higher rewards. Per definition, the normalized training reward lies in the range from 0 to 1 since $\min \leq f(x_i) \leq \max$. If the optimizer mistakes the greatest value for the minimum, it gets a reward of $r_i = \frac{\max - f(x_i)}{\max - \min} = \frac{\max - \max}{\max - \min} = \frac{0}{\max - \min} = 0$. In the case that the optimizer predicts the minimum value accurately, the reward is $r_i = \frac{\max - f(x_i)}{\max - \min} = \frac{\max - \min}{\max - \min} = 1$. This property is preserved independent of the actual “slice” the optimizer is working on. This normalized training reward allows an optimization algorithm to get high rewards close to one even when it is operating on a “slice” that contains only high objective function values in the global perspective. This is important as we want to be able to compare the performance of an optimization algorithm on an unfavorable “slice” to an optimization algorithm on a favorable “slice”. This way we judge the optimization algorithm rather than the circumstances it is exposed to.

We used this reward definition for the training and the evaluation. The chosen reward definition requires the knowledge of the values max and min for a given “slice” before executing the optimization algorithm. We approximated these as it would have been too time-consuming to find them by brute-force-search. We make our approximation clear with the help of an example. Let us assume that we operate on an objective function with six input dimensions and two free parameters. Let the constraints be $x_1 = a$ and $x_2 = b$ for two random values a and b. If the global minimum of the objective function lies at the location x*$= (0, 0, 0, 0, 0, 0)$, we took $x^{\text{approx}} = (a, b, 0, 0, 0, 0)$ as an approximation for the minimum value on the “slice” satisfying the constrains. We proceeded analogously with the maximum. Technically, we measure how accurately the optimization algorithms converge to the approximated minimum of an objective function on a given slice. However, the approach works just as well when defining the reward to be the negative objective function value $r_i = -f(x_i)$. This would lead to the same learning results because lower objective function values lead to higher rewards. On the other hand, this has the disadvantage that it is not apparent from the reward alone, if an optimization algorithm is performing poorly or if the “slice” it is operating on contains only high objective function values.

Mean Sqared Error

In addition to the normalized reward, we calculate the mean squared error between the approximated minima $f(x^{\text{approx}})$ and the predicted minima. The expression $f_i(x)$ represents the predicted minimum when the optimization algorithm starts at the i-th starting point.

$$\text{MSE} = \frac{1}{N_{\text{start}}} \cdot \sum_{i=1}^{N_{\text{start}}} (f_i(x) - f(x^{\text{approx}}))^2 \tag{4.12}$$

The minimal MSE is 0 whereas the maximal value is unbounded in principle. The higher the normalized training reward and the lower the MSE, the better the optimization algorithm is performing. A perfect optimization algorithm is one that always finds the approximation of the global minimum on a given slice with high accuracy. Such an optimizer converges to a normalized training reward of 1 and an MSE of 0, no matter which starting point it had. A miserable optimization algorithm is one that always considers the maximum to be the minimum, despite minimizing. This optimizer converges to a normalized training reward of 0.

4.2.8 External Libraries

In this section, we acknowledge external libraries that we make use of.

Tensorflow [MAP+15] and pytorch [PGM+19] are free open-source software libraries. They are the most used machine learning and artificial intelligence frameworks. The main focus of Tensorflow lies in the training and inference of deep neural networks. It was developed and released by Google. TensorFlow supports the programming languages Python, JavaScript, C++, and Java. We want to highlight three useful features TensorFlow offers:

- AutoGraphing - This functionality employs a Tensorflow computation graph. Executing this graph instead of python code often results in substantial performance gains.
- Eager execution - This is a mode in which statements are evaluated instantly rather than constructing a computational graph for later execution. This option makes it a lot easier to debug code.
- Optimizers - TensorFlow provides implementations for the most known optimizers needed for the training of neural networks, including Stochastic Gradient Descent (SGD), ADAM, and ADAGRAD.
- TF-Agents - In TensorFlow there are many on-policy and off-policy algorithms implemented already. Among others, there are standard implementations with default parameters for the REINFORCE- and the PPO-agent.

PyTorch was originally developed by Meta AI but is part of the Linux Foundation umbrella today. It mainly is used in the programming language Python, but there is an interface for C++ as well. We chose to use TensorFlow for our implementation as we are slightly more familiar with this framework and its application programming interface (API).

We use the helpful library NumPy [HMW+20] for handling large arrays and matrices of multiple dimensions. They also provide many useful high-level mathematical operations. For visualizing the results we got out of our experiments, we use the plotting library Matplotlib [Hun07].

Table 4.3 shows the versions we used for Python and all external libraries.

Library	Version
Python	3.8
TensorFlow	2.10.0
Matplotlib	3.5.3
NumPy	1.23.2

Table 4.3: Versions we used for external librarys

4.3 Limitations

In this section, we show the difficulties we encountered while conducting our work and how we dealt with them.

Our first implementation was runnable on Linux systems, but we noticed it was not on Windows systems. After intensive research and many rounds of trial and error, we found out that it has to do with the library “reverb” we used at that point. In this first version, we used “reverb” for implementing replay buffers. The latest version of the code, with which we conducted our experiments, does not have this issue anymore. We switched to using “TFUniformReplayBuffers” which are part of the TF-Agents library. It is possible now to run the code on Linux as well as on Windows. This hopefully increases the number of interested readers who are now able to run the code for themselves.

5 Experiments & Discussion

To rule out any performance differences due to the use of different hardware, we conducted all experiments on the same server. We used the server with the hostname "neon" which was made available to us by the Institute for Parallel and Distributed Systems at the University of Stuttgart. It is a Linux-based server with an Intel Haswell 72-Core processor and 504GB of RAM.

When we use the expression "one experiment", we refer to one run through the flowchart we discussed in Figure 4.1 for a fixed objective function with a fixed input dimension and a fixed number of free parameters. If not stated otherwise, we used the default parameter and hyperparameter values given in Table 4.1.

For every experiment we conducted, we evaluate the normalized training reward and the MSE. We introduced both in Subsection 4.2.7. We conducted experiments for the eight objective functions Ackley, Griewank, Levy, Rastrigin, Rosenbrock, Sphere, Styblinski-Tang, and Zakharov which we defined in Subsection 4.2.3. For every objective function, we conducted experiments for two, four, six, eight, and ten input dimensions. We chose even input dimensions because we had the number of free parameters $N_{\text{iteration}}$ in mind. This will become clearer in the next few lines.

Let us suppose, we have an objecitve function f with input dimension d. Theoretically, 0 to d of those input parameters could be $free$ parameters, e.g. constrained parameters. The two extreme cases $N_{\text{iteration}} = 0$ and $N_{\text{free}} = d$, however, are not interesting for us. As the title of this work indicates, we investigate "Parameter-dependent self-learning optimization". This implies that we cover functions including some parameter-dependencies and we want optimizable functions. Parameter-dependencies describe parameters for which some conditions must be satisfied. In the case of $N_{\text{free}} = 0$, we would optimize a function without any parameter-dependencies. Further, if all input parameters are fixed by constraints, as is the case for $N_{\text{free}} = d$, there would be no parameters left to optimize. In this case, the objective function f has fixed inputs and is therefore a fixed constant itself. For this reason, we only consider 1 to $d - 1$ as possible values for N_{free}.

We want to get a representative overview of the behavior of a learned optimization algorithm for a specific objective function f and input dimension d for different numbers of free parameters N_{free}. Therefore, we picked three values N_{free} to discover trends. We picked the lowest possible value $N_{\text{free}} = 1$, the middle option $N_{\text{free}} = \frac{d}{2}$ and the highest possible value $N_{\text{free}} = d - 1$. To ensure that $\frac{d}{2} \in \mathbb{N}$, we picked only even numbers for the input dimensions d.

For the special case $d = 2$, it happens to be that $1 = \frac{d}{2} = d - 1$. This means, for $d = 2$ we only conducted one experiment per objective function. In all other cases, namely for $d \in \{4, 6, 8, 10\}$, we conducted three experiments per objective function. In total, this results in $N_{\text{2D experiments}} + N_{\text{objective functions}} \cdot N_{\text{input dimensions}} \cdot N_{\text{free parameters}} = 8 + 8 \cdot 4 \cdot 3 = 8 + 96 = 104$ "basis" experiments.

Strictly speaking, agents are the ones receiving rewards. Agents update their policies which represent optimization algorithms. In the following, we will use the terms "agent", its respective "policy" after a certain number of training iteration steps and the represented "optimization algorithm" interchangeably.

5.1 Two Input Dimensions

We combined all of our 2D results in the two graphics in Figure 5.1. The left graphic shows the normalized training reward against the number of training iteration steps. On the right, we illustrated the logarithmic squared error against the number of training iteration steps. In the latter diagrams, solid lines represent MSEs and the transparent areas around them show the standard deviation resulting from different starting positions. For more details about the evaluation, we refer to Subsection 4.2.7. This evaluation allows us to find differences and commonalities between the learned optimization algorithms on the different objective functions. The x-Axis shows the number of training iteration steps the optimization algorithm was trained for. We decide to use a logarithmic y-Axis on the vast majority of MSE graphics as this allows us to clearly identify subtle differences. The legend is located in the lower right corner for reward-plots and the upper right corner for MSE-plots, as we would expect those areas to be the ones where the least lines pass through.

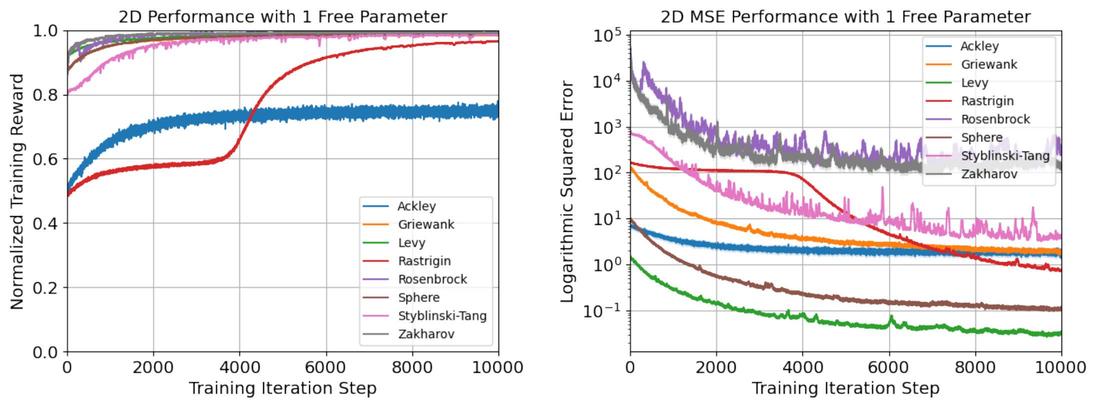


Figure 5.1: 2D performance and MSE with one free parameter on the eight objective functions

In both graphics, the legend shows which color is assigned to which objective function. For example, the blue line shows the performance of the learned optimization algorithm on the 2D Ackley function with one free parameter, dependent on the number of training iteration steps it was subjected to.

We will analyze these first two diagrams in slightly more detail as the observations and conclusions we can make in these are mostly transferable to the following experiments.

5.1.1 Normalized Training Reward

The initial normalized training reward is higher for some objective functions than for others. For example, the optimization algorithm operating on the Rastrigin function has an initial normalized training reward of approximately 0.5 (red line) whereas the one operating on the Styblinski-Tang function has an initial normalized training reward of approximately 0.8 (pink line). This is partly due to the different codomains of the objective functions. We defined the normalized training reward to be $r = \frac{max-f(x)}{max-min}$. This fraction converges to one for huge values of max, as the influence of $f(x)$ and min on r shrink. This means optimization algorithms for objective functions with big codomain tend to get normalized training rewards closer to one from the beginning. Another reason for optimization algorithms operating on different objective functions starting with different initial normalized training rewards is the differences in the properties of the underlying objective functions. For instance, most inputs from the domain of the Ackley function result in an objective function value $f(x)$ near the maximum value max. In contrast, the Zakharov function has a very wide area where the objective function value $f(x)$ is relatively close to the actual minimum value min. This means an optimizer with no experience will get a greater normalized training reward for the Zakharov function than for the Ackley function just by randomly guessing.

It is also noteworthy that the blue line corresponding to the Ackley function is thicker than the others. A thick line means that the normalized training reward is more volatile and fluctuates more from one training iteration step to the next. Such fluctuations appear due to local minima and maxima causing changes in the objective function value $f(x)$ which affects the normalized training reward r. The reason for the blue line being thicker is that the extent to which a small change in the input can cause a big jump in objective function value relative to the value max is greater for the Ackley function than for other objective functions.

We can see that the normalized training reward is increasing for higher numbers of training iteration steps, independent of the objective function. However, the lines are not monotonously growing rather they do have small disruptions. This was expected because a higher number of training iteration steps means that the reinforcement learning algorithm has passed more training loops. In every training loop, the agent collects experience and adapts better to the objective function it is working on. Therefore, an agent who has trained longer gathered more experience and can estimate the location of minima more precisely. This fact results in a higher normalized training reward. Small disruptions in the normalized training reward are ignorable as they do not affect the "learning process" in the long-term.

All optimization algorithms converge to a normalized training reward close to the value one within 3000 training iteration steps except for the one dealing with the Rastrigin function (red line) and the one dealing with the Ackley function (blue line). Differences like this emphasize the importance of studying various objective functions to capture as many cases as possible.

The red line runs uniquely compared to the other lines. If we ran the reinforcement learning algorithm for $N_{\text{iteration}} = 3000$ we would have thought that the red line converges to a normalized training reward just above 0.6. However, after about 3500 training iteration steps the red line began to rise again. Ultimately, the red line comes very close to a normalized training reward of one after $N_{\text{iteration}} = 10000$ training iteration steps. Such radical changes can be explained by the agent changing its optimization strategy. One possible trigger for a change in optimization strategy is when agents discover an action which they never tried before. Agents discover such actions due to the non-deterministic nature of the collect-policy. The collect-policy is the policy which is used for collecting experience. Collect-policies have built-in randomness which ensures that the agent explores the domain occasionally rather than only exploiting the options he considers to be most promising.

The blue line converges to a normalized training reward of approximately 0.75. After observing the sudden climb of the red line after $N_{\text{iteration}} = 3500$, we wondered, if more training iteration steps would lead to the blue line also converging to a normalized training reward of one. We repeated the experiment for the 2D Ackley function with one free parameter and $N_{\text{iteration}} = 100000$ training iteration steps but this did not further improve the performance. The optimization algorithm operating on the Ackley function (blue line) converges to a value smaller than one since it got stuck in one of the several local minima of the Ackley function and mistakes it for the global minimum for a given slice. The objective functions we chose are purposely difficult for optimization algorithms in the sense that it is difficult for the optimization algorithms to find the actual minimum.

5.1.2 Mean Squared Error

Analogous to the normalized training reward, the MSE is at the beginning higher for some objective functions than for others. This was expected because optimization algorithms learned by RL behave like random guessers without any experience. The chances of picking values far from the actual minimum are higher when the objective function has a large codomain.

As expected, the MSE decreases with an increasing number of training iteration steps, no matter which objective function is trained on. This shows that the optimization algorithms become better at estimating where to find the minimum on a given slice with more experience. However, there are specific differences, depending on which objective function was trained on. For example, the blue line (Ackley) started with an MSE of about ten and converges to an MSE of approximately one, while the orange line (Griewank) which had an initial MSE of about 150 also converges to an MSE of one in the end. This means that the agent for the Griewank function improved more after $N_{\text{iteration}} = 10000$ in comparison to the beginning than the agent on the Ackley function.

All MSE-lines approach an MSE below ten after $N_{\text{iteration}} = 10000$ training iteration steps besides the lines for the Rosenbrock function (purple line) and the Zakharov function (grey line). It is no coincidence that these are the two objective functions which have by far the biggest domains in our collection.

It was foreseeable that the curve for the Sphere function (brown line) reaches a very low MSE as the Sphere function is by far the easiest objective function of our collection to optimize since it has no minima besides the global one. Further, it has a comparably small codomain. Surprisingly, the optimization algorithm operating on the Levy function (green line) achieves an even lower MSE for $0 \leq N_{\text{iteration}} \leq 10000$. The reason for this can be found in the structure of the Levy function by looking at Figure 4.2. When observing the 2D Levy function with one free parameter, we examine slices for fixed values of x_1 (called x in Figure 4.2). The objective function value of the 2D Levy function is nearly the same for a given x. Therefore, the optimization algorithm operating on the 2D Levy function with one free parameter performs well, concerning both normalized training reward and MSE.

The MSE of the optimization algorithm on the Rastrigin function (red line in the right graphic) starts to rapidly decline at the exact training iteration step for which the normalized training reward of the optimization algorithm on the Rastigin function (red line in the left graphic) starts to grow. When the normalized training reward $r = \frac{max - f(x)}{max - min}$ grows for a fixed objective function, this implies that $f(x)$ must have become smaller and a smaller f leads to a smaller MSE. This is the case because a fixed objective function has fixed values for max and min and the only component left to change in the reward definition is the objective function value $f(x)$. Nonetheless, it is a misconception to think that a higher reward implies a smaller MSE in general. For example, in the left graphic, the red normalized training reward (Rastrigin) is below the purple normalized training reward (Rosenbrock) for all training iteration steps. Despite this fact, in the right graphic, the red MSE is below the purple MSE for all training iteration steps. The right formulation would be: A higher reward implies a smaller MSE for a fixed triple of an objective function, an input dimension, and the number of free parameters!

5.2 Higher Input Dimensions

Figure 5.2 and Figure 5.3 show the plots of normalized training reward and MSE, respectively, for a selection of the "basis" experiments including input dimensions four and ten for the objective functions Ackley, Rastrigin, Sphere and Styblinski-Tang. The experiments covering the remaining input dimensions and objective functions can be found in Figure B.1 and Figure B.2 in the appendix. We grouped optimization algorithms by objective function and input dimension. The diagrams show their behavior for a fixed function and input dimension with a changing number of free parameters N_{free}. In these diagrams, different colors indicate a different N_{free}. The legend shows which color is assigned to which N_{free}.

We notice that the graphs for the same objective function (the graphs in a row) look very similar to each other. That is because changing the dimensionality of an objective function does not change its properties. This leads to agents learning similar strategies and this leads to comparable progressions of the normalized training reward.

Further, the order of the lines from good (high reward and low MSE) to bad (low reward and high MSE) is genarally $N_{\text{free}} = d - 1$ (green line), $N_{\text{free}} = d/2$ (orange line) and $N_{\text{free}} = 1$ (blue line). This indicates that the higher the number of free parameters N_{free}, the easier it is for a learned optimization algorithm to find the global minimum on a slice. In other words: The higher the number of optimization parameters $N_{\text{optimization}} = d - N_{\text{free}}$ the more difficult it is to optimize an objective function. According to this, the greater action space following from a higher $N_{\text{optimization}}$ has more influence on the difficulty of optimizing an objective function than the increasing combinatorial "slice-possibilities" which result through more free parameters. We refer to Section 4.2.4 for a more in depth explaination of the influence of N_{free} and $N_{\text{optimization}}$ on the action space and the number of "slice-possibilities".

After a certain number of training iteration steps, all diagrams, except for the Rastrigin ones, show that the optimization algorithms on different numbers of free parameters end up converging to a very similar normalized training reward. Through deeper inspection of the graphs of the Rastrigin functions one could imagine that if trained for more training iteration steps, the different lines eventually converge to nearby values as well. This is especially remarkable in the digram showing the 4D Rastrigin performance to the left. We observed that the conspicuous "turning point" for the 4D Rastrigin function with two free parameters is found after $N_{\text{iteration}} = 6500$ steps, almost 3000 training iteration steps later than the "turning point" for the line showing three free parameters. We wondered if the "turning point" shifts to the right with fewer free parameters and a fixed input dimension for the Rastrigin function. We investigate this question in Section 5.5.

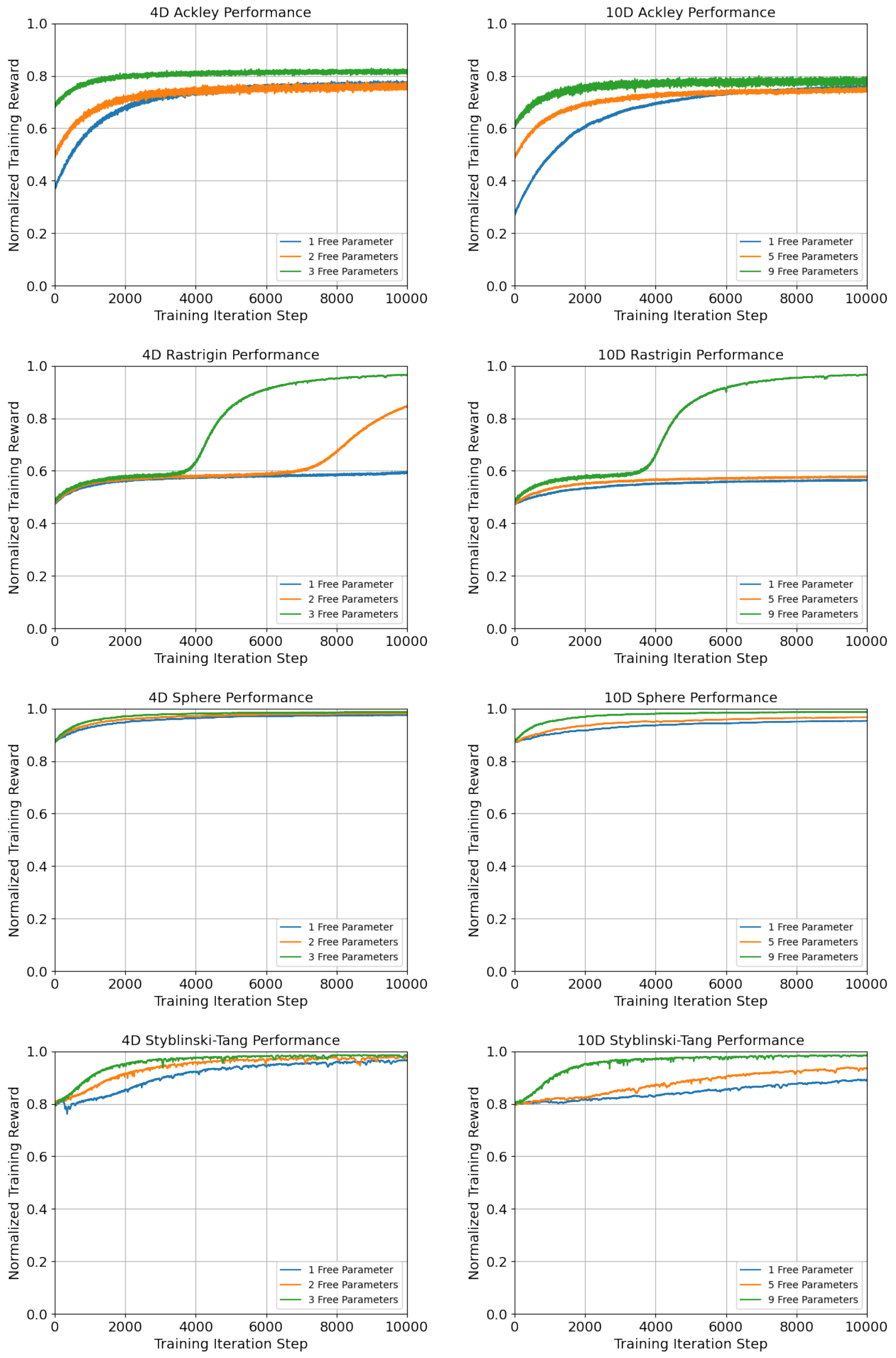


Figure 5.2: 4D and 10D parameter-dependent performance on Ackley, Rastrigin, Sphere and Styblinski-Tang

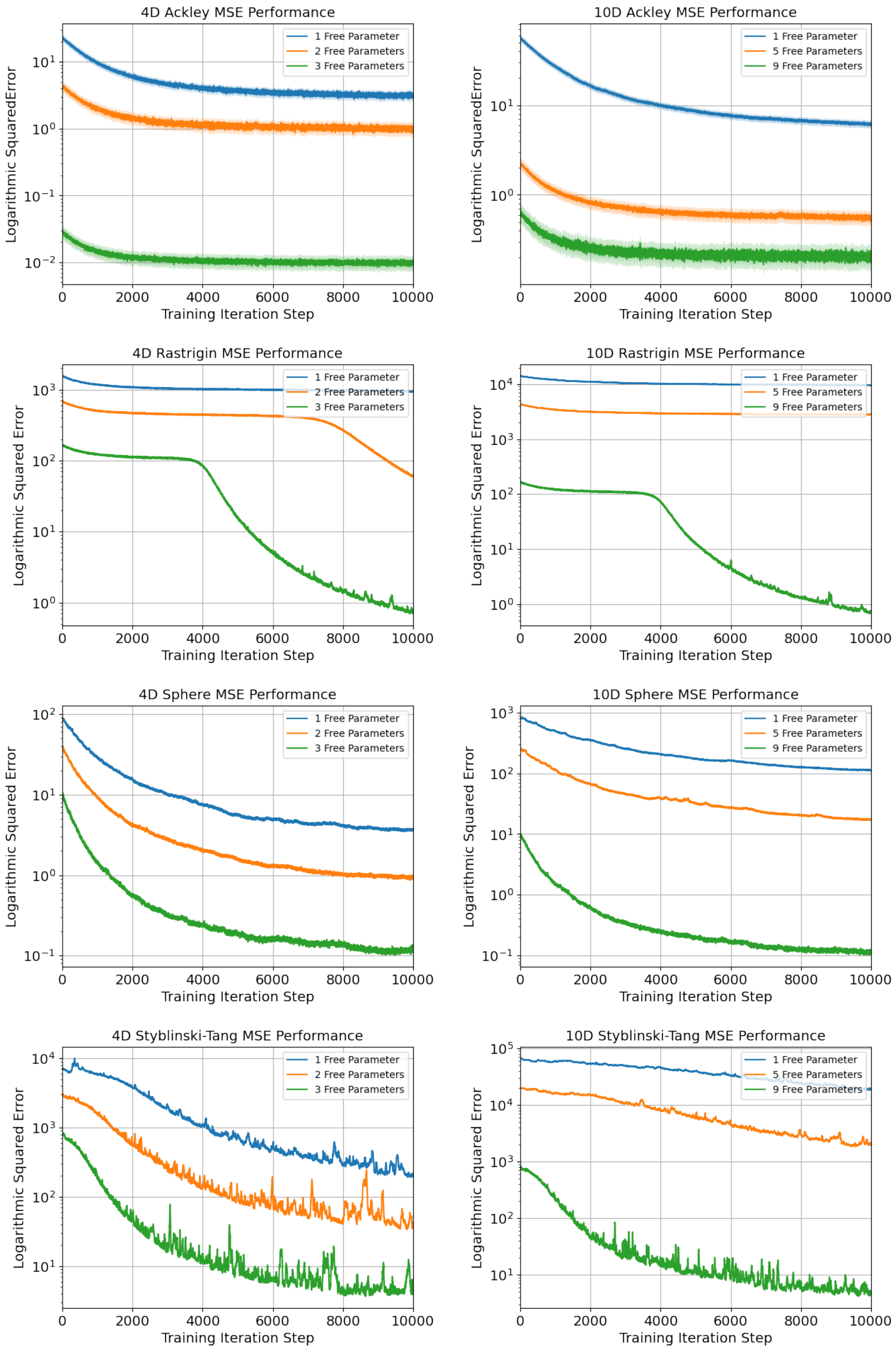


Figure 5.3: 4D and 10D parameter-dependent MSE on Ackley, Rastrigin, Sphere and Styblinski-Tang

5.3 One Free Parameter

The following two figures, namely Figure 5.4 and Figure 5.5, are based one the same experiments we reviewed so far. However, we changed the way we grouped different experiments. We compare how optimization algorithms perform on the same objective function with one free parameter for different input dimensions. Different input dimensions are represented with different colors.

We observe that all lines for a specific objective function converge to almost the same normalized training reward. Despite this fact, the MSEs differ for different input dimensions on the same objective function with one free parameter. We recognize an order between the input dimensions, independent of the objective function. The lower the input dimension, the lower the MSE for $N_{\text{free}} = 1$. This was predictable because optimizing a 10D objective function with one free parameter means there are nine optimization parameters left. This is a much more demanding task for a learned optimization algorithm than optimizing a 2D objective function with one free parameter and one optimization parameter. This results in optimization algorithms performing better and getting lower MSEs for lower input dimensions.

Rastrgins “turning point” is only visible for $d = 2$ (blue line) for $0 \leq N_{\text{iteration}} \leq 10000$. In Section 5.5 we see that the “turning point” becomes visible for 4D Rastrigin with one free parameter beyond 10000 training iteration steps.

When looking closely one discovers that the Styblinski-Tang function deviates from the previously mentioned order in the plot showing the normalized training reward. The explanation for this is the big drop in normalized training reward after around 500 training iteration steps. It is well-known that instabilities in the optimization strategy can occur in RL, in particular at the beginning of the training process. However, it is evident that the agent recovers quickly from this “irritation” and is not influenced by this in the long-term when looking at the MSE-graph. The MSE of the optimization algorithm with input dimension eight (red line) recovered shortly after the breakout and drops below the MSE of the optimization algorithm on input dimension ten (purple line).

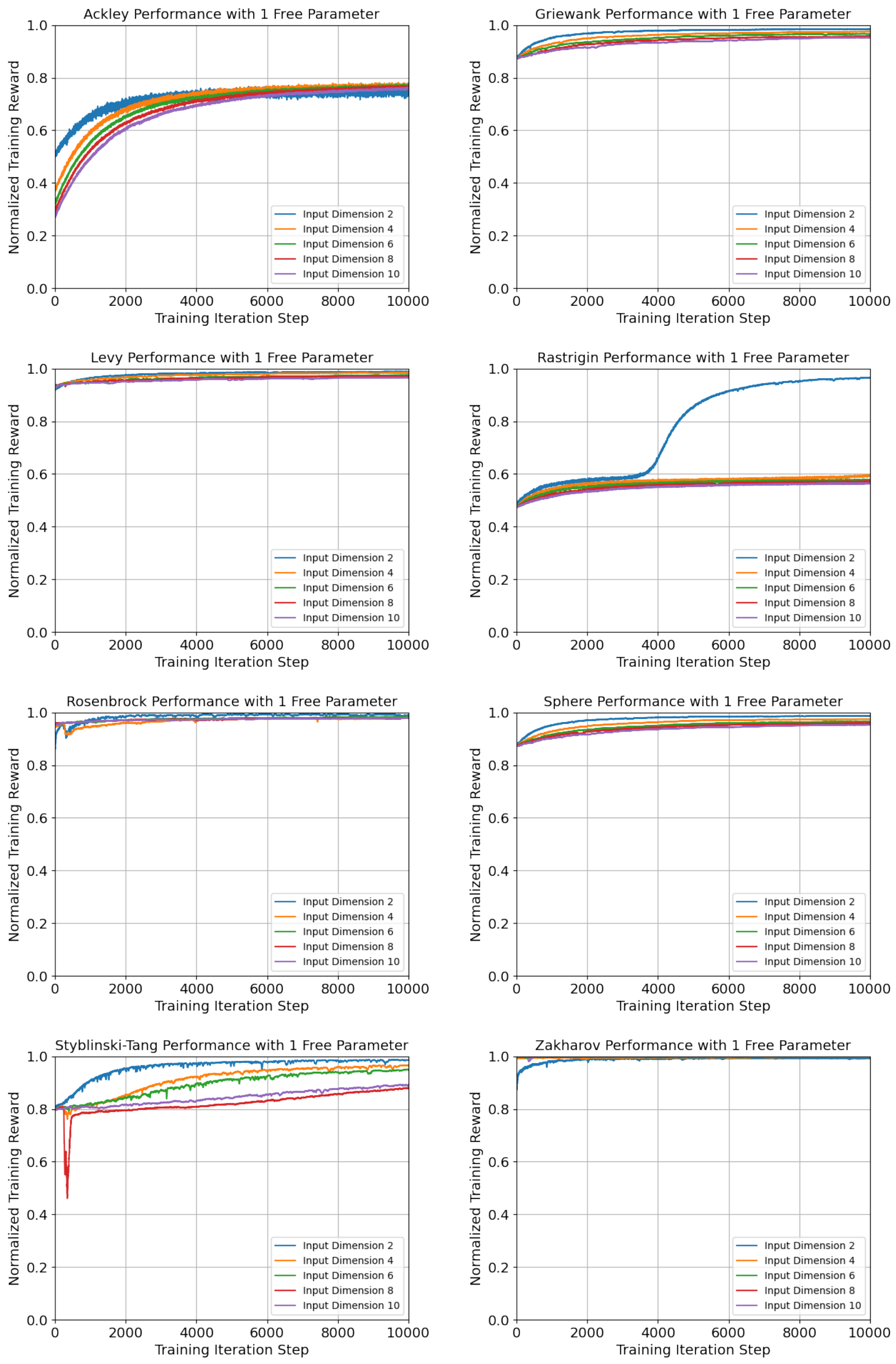


Figure 5.4: One free parameter performance for input dimensions 2-10 on the eight objective functions

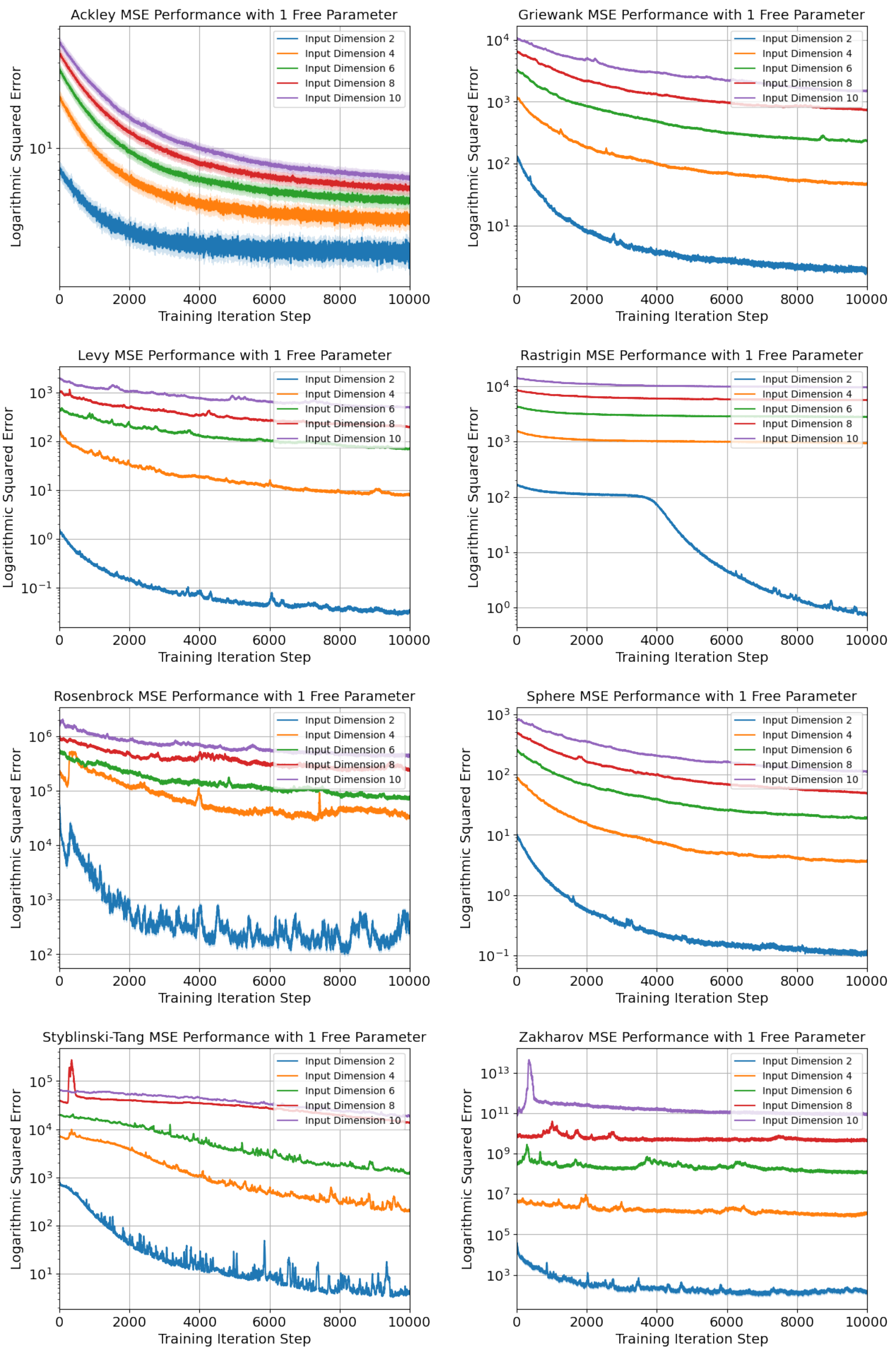


Figure 5.5: One free parameter MSE for input dimensions 2-10 on the eight objective functions

5.4 D-1 Free Parameters

After comparing optimization algorithms operating on the same objective function with one free parameter for different input dimensions, we now compare optimization algorithms with $d - 1$ free parameters. Figure 5.6 shows the normalized training reward and Figure 5.7 shows the Logarithmic Squared Error.

The graphs visualizing the normalized training reward show that the lines, representing different input dimensions d with $N_{\text{free}} = d - 1$, run very close to each other. Only the Ackley-plot allows for a clear distinction between the lines. The graphics for the MSE show that the almost identical normalized training rewards come along with not distinguishable MSEs. The exceptions are the Ackley- and the Zakharov-plot. They show the trend that a higher input dimension leads to a lower normalized training reward and higher MSE. This is logical since more free parameters mean there are more “slice-possibilities”. Therefore, it should be harder to optimize with more free parameters because the optimization algorithm has to generalize over more possible configurations of free parameter values.

Interestingly, this trend is not visible in the other six diagrams. According to those, a learned optimization algorithm has the same difficulties when optimizing a 2D objective function with one free parameter or a 10D objective function with nine free parameters. This can be made plausible by noticing that $d - 1$ free parameters imply that there is only one optimization parameter left. According to this, all optimization algorithms, whether it is on 2D with one free parameter or 10D with nine free parameters, have the same action-space.

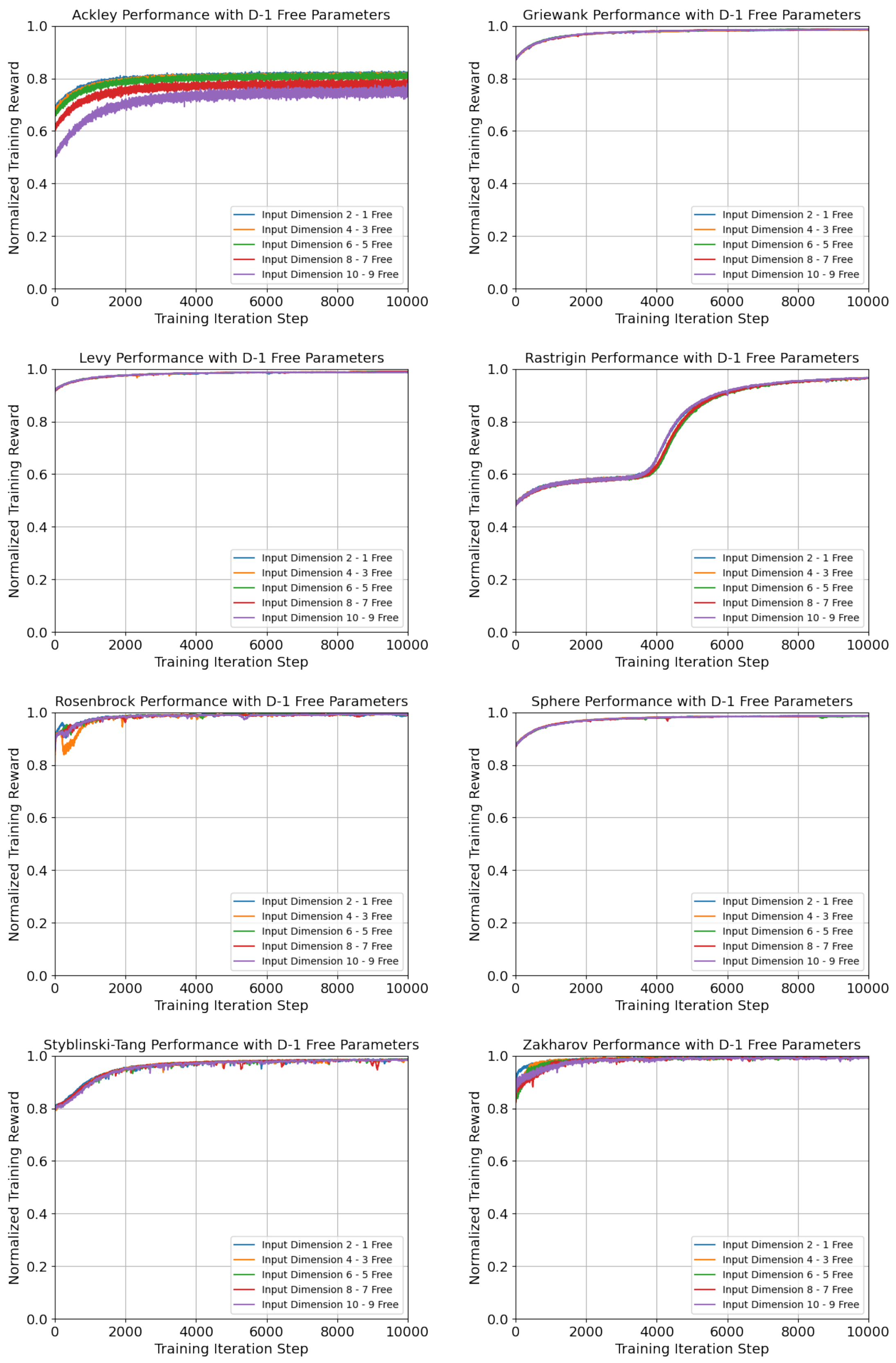


Figure 5.6: D-1 free parameters performance for input dimensions 2-10 on the eight objective functions

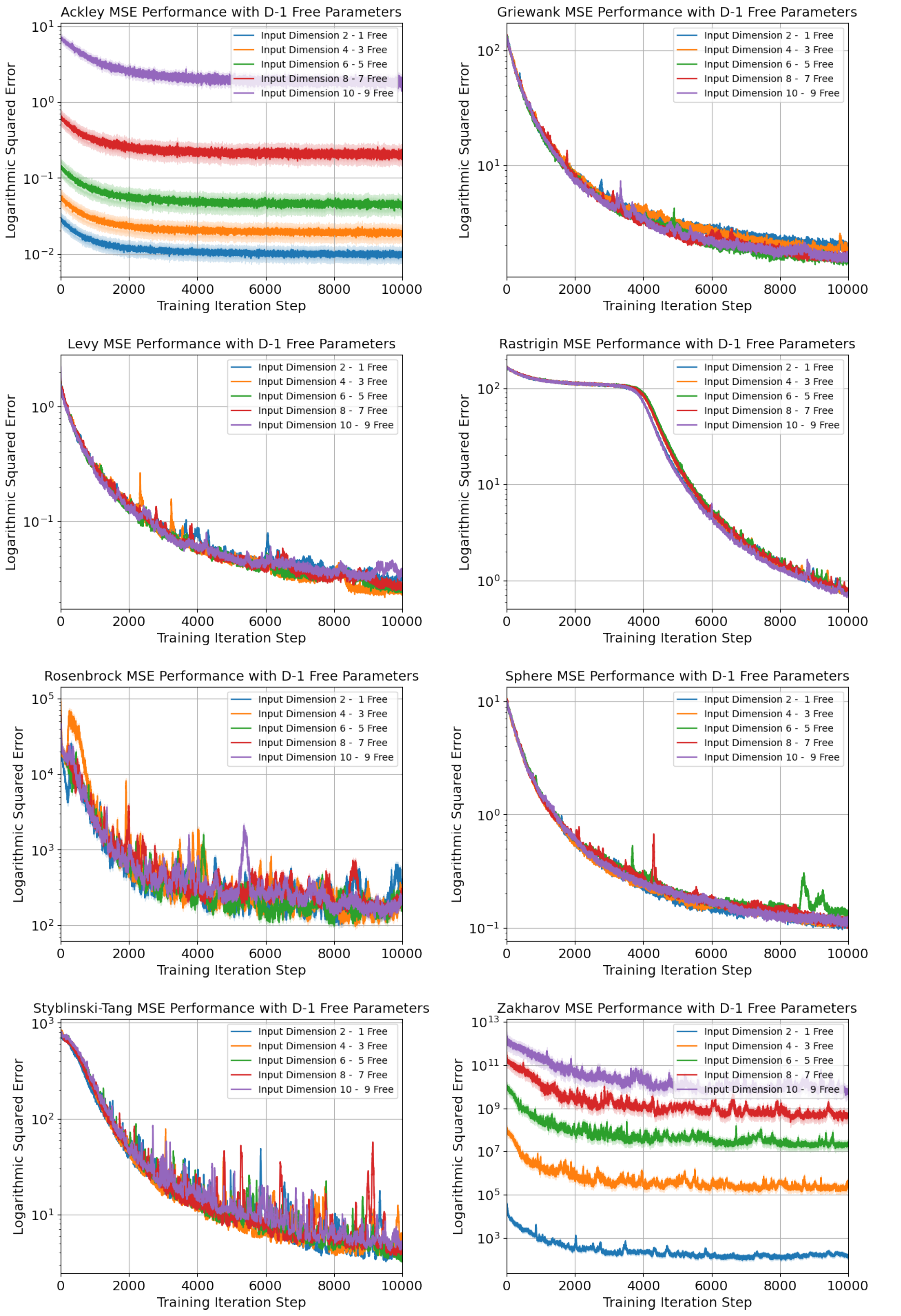


Figure 5.7: D-1 free parameters MSE for input dimensions 2-10 on the eight objective functions

5.5 Long Training

After conducting our main experiments we were curious how the increased number of training iteration steps might influence the performance of learned optimization algorithms. The experiment which produced the best improvements is shown in Figure 5.8.

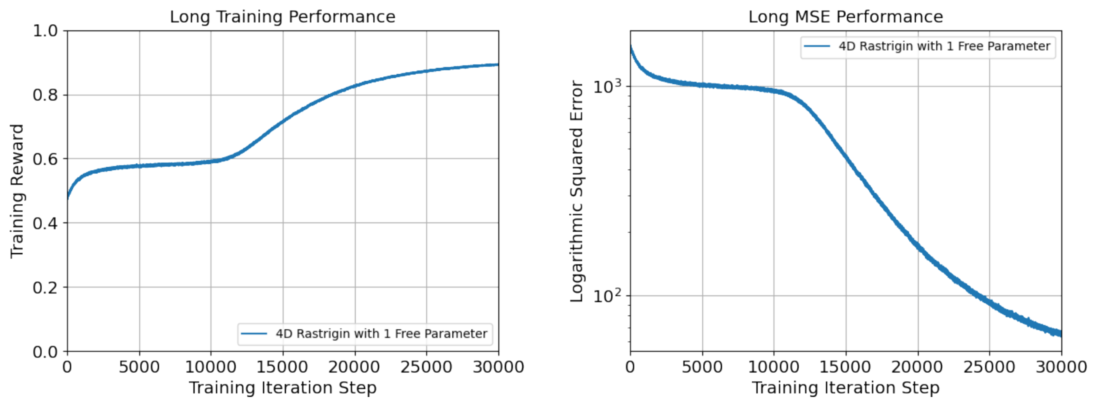


Figure 5.8: Long training performance and MSE on the 4D Rastrigin function with one free parameter

The figures show the behavior of an optimization algorithm operating on the 4D Rastrigin function with one free parameter. As already mentioned, when only looking at $0 \leq N_{\text{iteration}} \leq 10000$, it seems that the normalized training reward converges to a value of roughly 0.6 and the MSE converges to 10^3. However, as these graphics show, continuing the training for another 20000 training iteration steps revealed the real long-term trend. After a total of $N_{\text{iteration}} = 30000$ steps, the normalized training reward reaches 0.9 and the MSE drops significantly. Therefore, training for more training iteration steps has a tremendous impact on the performance of the optimization algorithm on the 4D Rastrigin function with one free parameter. We assume that more training iteration steps are highly beneficial to optimization algorithms which do not show a “turning point” on the Rastrigin function. In our opinion, it is likely that every optimization algorithm operating on the Rastrigin function shows a “turning point”, no matter what the actual combination of input dimension and number of free parameters is. We infer that the “turning point” shifts to the right for higher input dimensions and fewer free parameters. This hypothesis has yet to be proven.

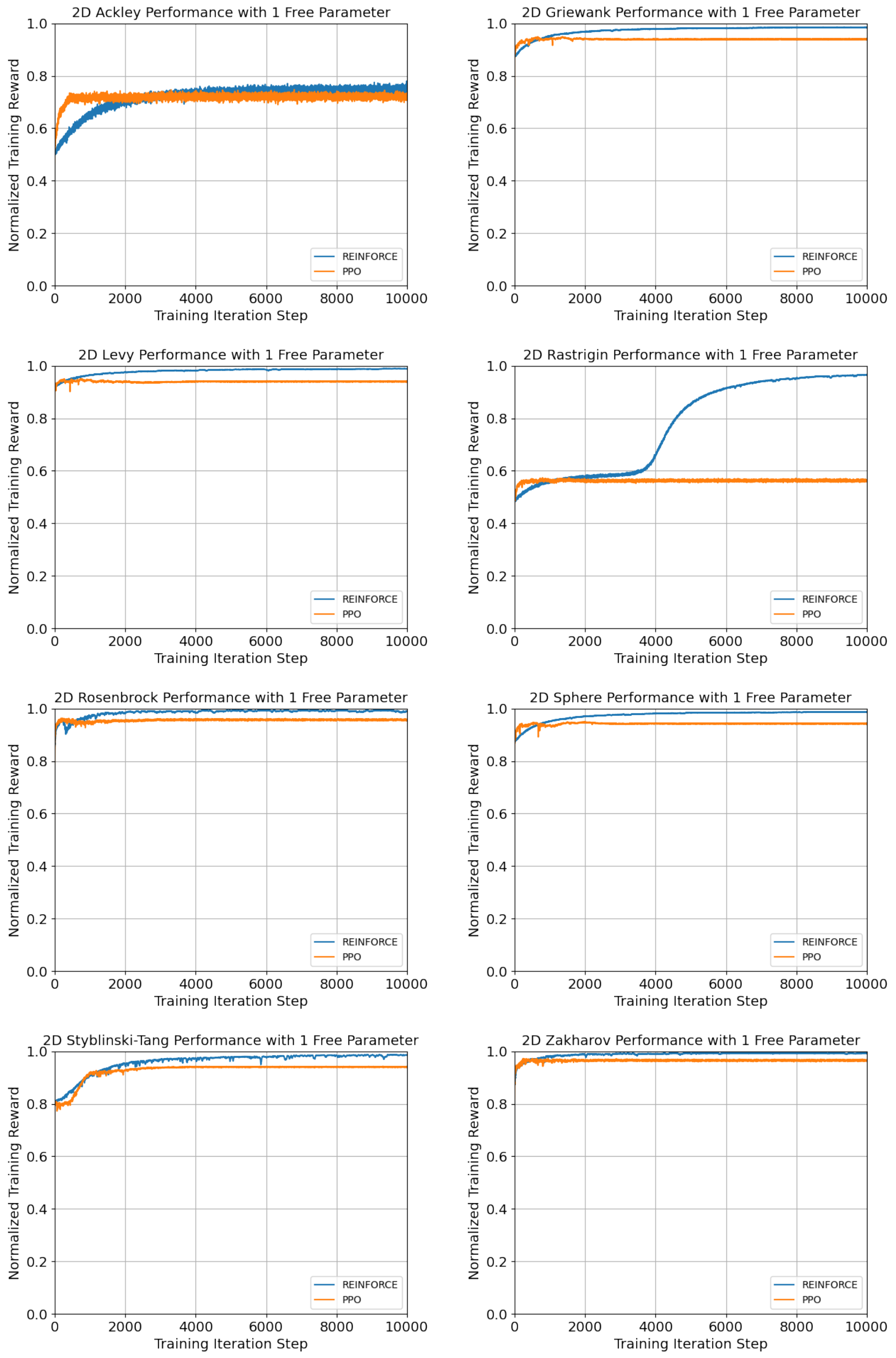


Figure 5.9: REINFORCE and PPO performance on the eight objective functions in 2D

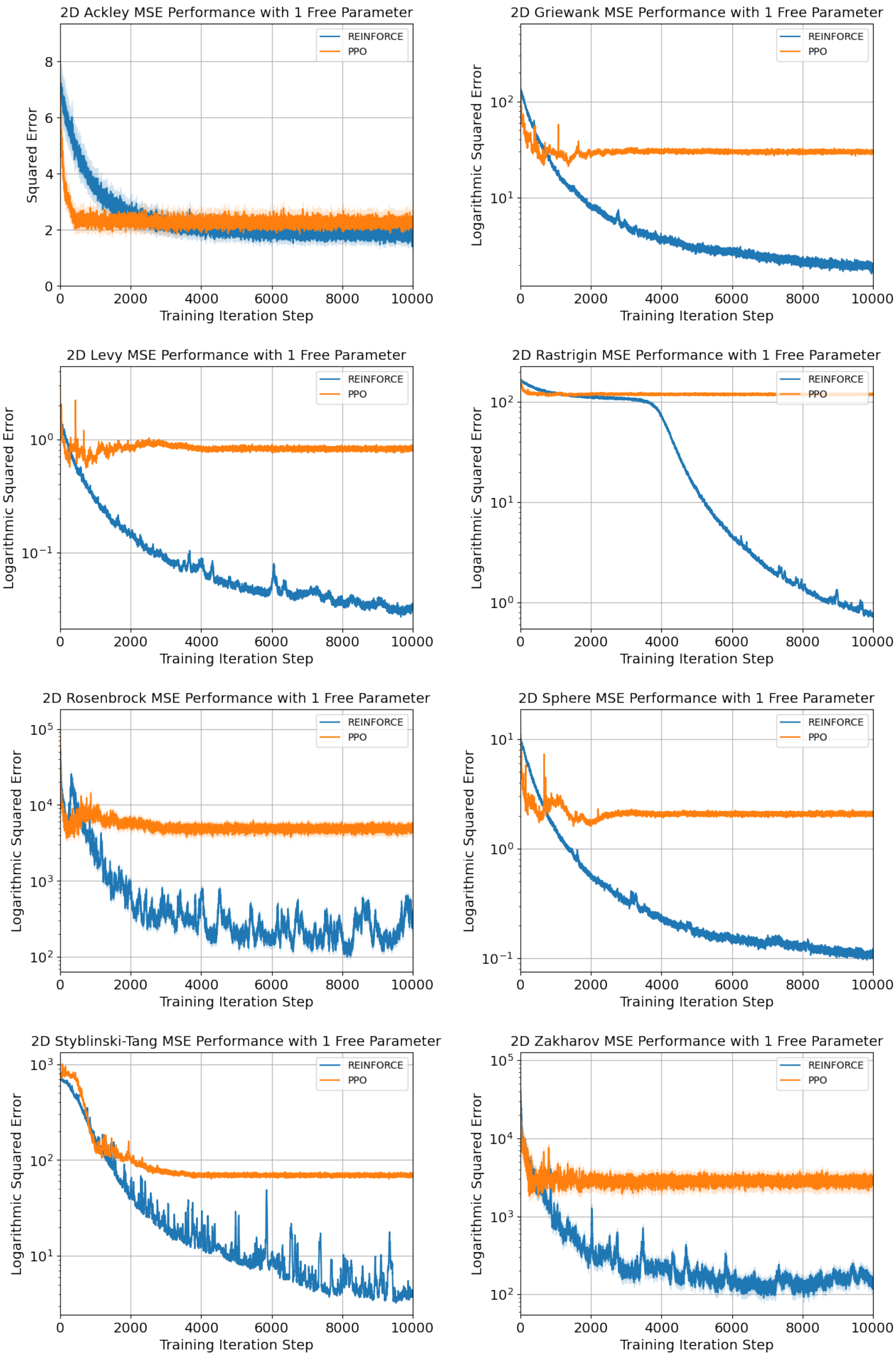


Figure 5.10: REINFORCE and PPO MSE on the eight objective functions in 2D

5.6 REINFORCE and Proximal Policy Optimization

In this section, we try to find out whether the REINFORCE- or the PPO-algorithm is preferable to the other in our setting. The normalized training reward can be seen in Figure 5.9 and the MSE-plots can be found in Figure 5.10. We focus on 2D functions as the runtime “exploded” when using the PPO-agent on higher dimensions. We analyze the runtime of all conducted experiments in Section 5.8.

The plots showing the normalized training reward let us assume that PPO-agents reach their final normalized training reward faster. This is also visible in the MSE-plots as the orange line (PPO) starts to fall steeper than the blue line (REINFORCE) in the beginning. However, in all presented cases, the blue line surpasses the normalized training reward of the orange line, resulting in lower MSEs for the optimization algorithms based on the REINFORCE-algorithm.

Thus, PPO-agents should be preferred when training for a low number of training iteration steps of under $N_{\text{iteration}} = 1000$. On the other hand, the REINFORCE-agent outperforms the PPO-agent concerning the normalized training reward and the MSE in the long-term for all of our chosen objective functions.

5.7 Neural Networks and Recurrent Neural Networks

Here we compare the differences between using feedforward NNs and RNNs with LSTMs. We introduced those in Section 3.3. Figure 5.11 and Figure 5.12 display the results we got.

We limited these experiments to the input dimension $d = 10$ because we are mostly interested in the effects which take place in higher dimensions. We looked at one free parameter (left column) and nine free parameters (right column). The blue line represents optimization algorithms learned using NNs and the orange line represents ones which used RNNs with the extension of LSTMs. All graphs, the ones showing the normalized training reward as well as the ones showing the MSE, make clear that using RNNs with LSTMs does not make any difference for our use case. The optimization algorithms progress nearly identically.

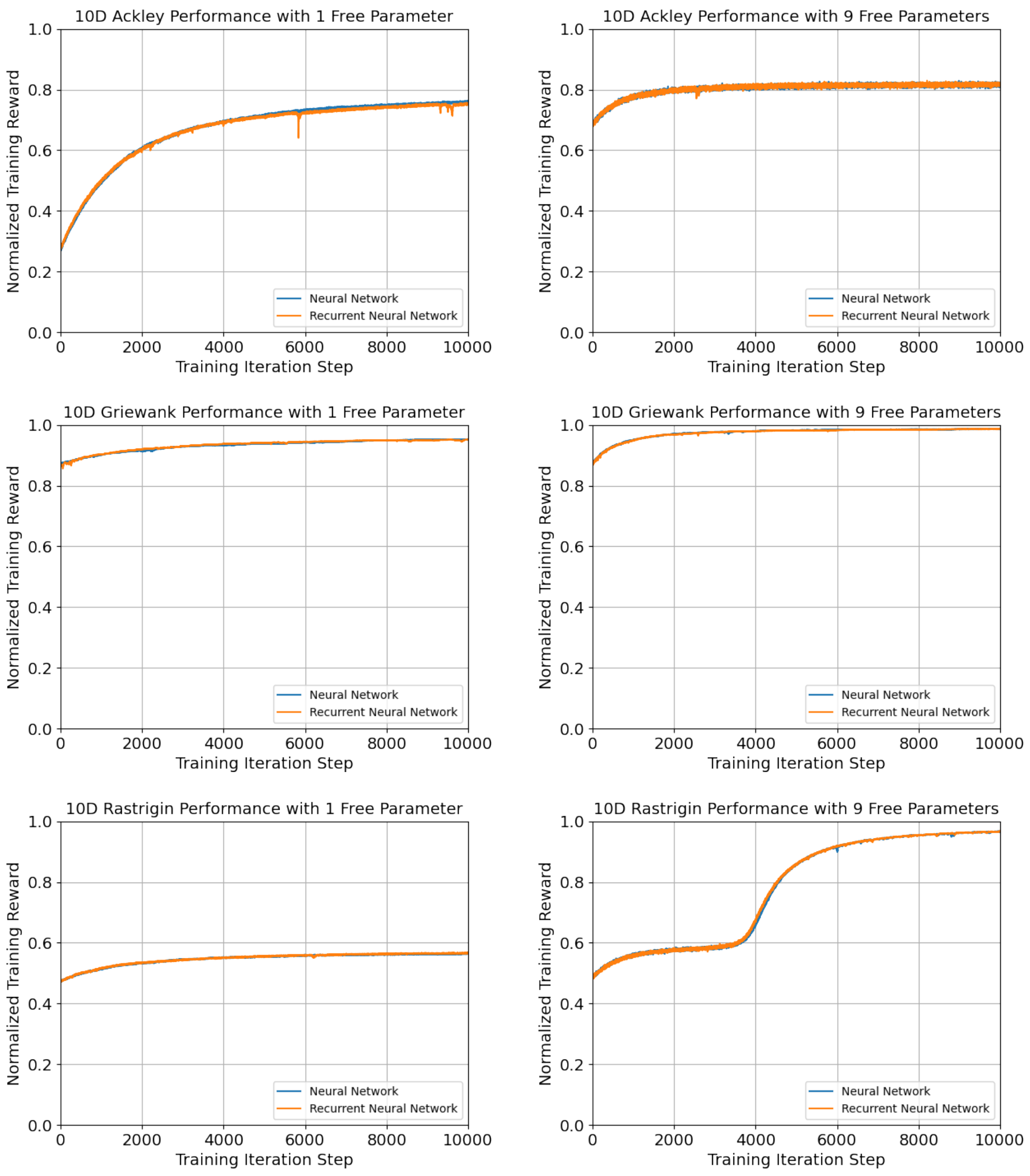


Figure 5.11: NN and RNN performance on Ackley, Griewank and Rastrigin in 10D with one and nine free parameters

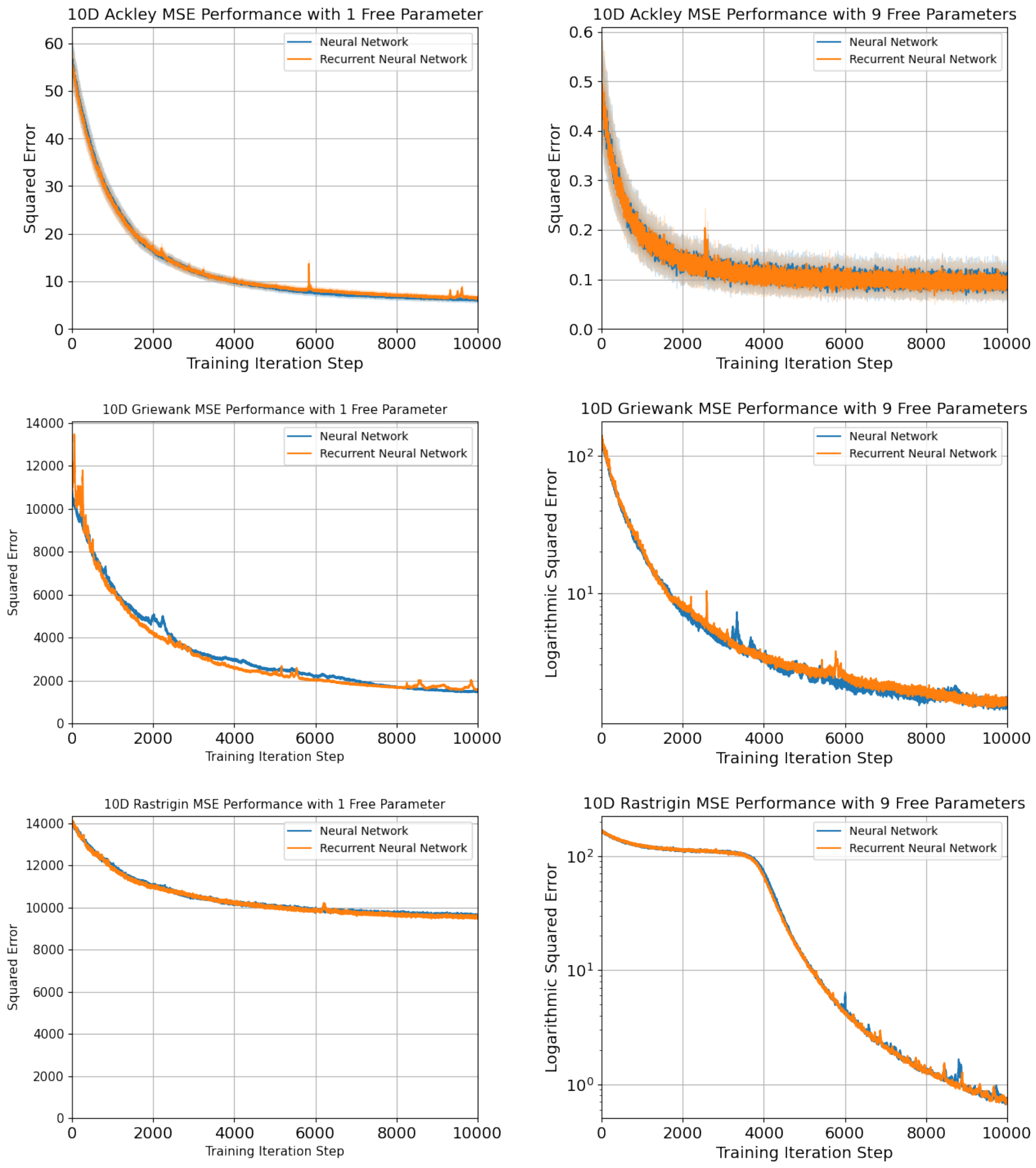


Figure 5.12: NN and RNN MSE on Ackley, Griewank and Rastrigin in 10D with one and nine free parameters

5.8 Runtime

This is the last section of our experiments. While conducting our experiments we measured the runtime. Optimization algorithms using the REINFORCE-algorithm and a feedforward NN have practically identical runtime for a fixed input dimension. Neither the objective function nor the number of free parameters matters. Table 5.1 shows the average runtime for an experiment in dependence on the input dimension. Increasing the input dimension from two to eight did not affect

Input Dimension	Runtime
2	35min
4	38min
6	41min
8	47min
10	101min

Table 5.1: Average runtime of one experiment for all input dimensions

the runtime drastically. However, experiments conducted with input dimension ten took more than twice as long in comparison to experiments with $d = 8$.

Conducting experiments for a higher number of training iteration steps resulted in linear growth of runtime. For example, the training on the 4D Rastrigin function with one free parameter for $N_{\text{iteration}} = 10000$ lasted about 38 minutes, whereas about 113 minutes were needed for $N_{\text{iteration}} = 30000$.

Using RNNs with LSTMs instead, or using the PPO-algorithm makes a difference in runtime as well. On average, the experiments we conducted with RNNs and $d = 10$ lasted 150 minutes. That is 1.5 times as long as with feedforward NNs. The experiments we conducted with the PPO-algorithm on two-dimensional functions took 71 minutes instead of 35 minutes with the REINFORCE-algorithm. This is a doubling in runtime. We started experiments with the PPO-agent on objective functions with $d = 10$ but we aborted them after finishing the first experiment as it needed over ten hours and 40 minutes to terminate.

6 Future Work

In this chapter, we propose some suggestions for continuing this line of work. We provide an outlook of the direction in which research based on this thesis might develop.

6.1 Further Experiments

The implementation as it currently is can be used for further interesting experiments without any modification.

One could examine the transferability of our learned optimization algorithms to unknown objective functions. It would be worth researching whether an optimization algorithm that was trained on one objective function performs well on other objective functions too. This could be possible for objective functions that share similar properties. This ability was already investigated in the bachelor thesis of Kilian Schüttler [Sch22]. However, in their approach no free parameters are integrable, and they restricted their work to $d = 2$. The behavior of this property is unclear when integrating free parameters. In addition, it is uncertain, how well transferability scales in high-dimensional setups.

Additionally, a deeper inspection of the experiments we conducted could reveal more insightful information. While training an agent on an objective function we saved checkpoints of the replay buffer and the learned policy in regular intervals. One could analyze the actions chosen by the policy with respect to the number of steps trained. Strategy changes may be recognizable.

6.2 Modifications

Another approach to extending our work is to change choices where we had to decide on one of several possible options. It would be worthwhile to analyze the influence of various changes to our study design.

Currently, we defined the normalized reward r in dependence on the last action a. Another option for defining the reward is to compute a relative notion we call “relative reward”. Lets assume we have the values x_{free} for the “free” parameters. We define x_i to be the concatencation of the “free” parameter values and the chosen action a_i in step i. For the first action a_1 we compute and save the resulting objective function value $f(x_1)$ as usual, but we set the first reward to be $r_1 = 0$. For all following actions a_i with $i \in \{2, \dots, N_{\text{episode}}\}$ we compute and save the resulting objective function value $f(x_i)$ again, but we define the reward to be $r_i = \gamma^{i-1} \cdot (f(x_{i-1}) - f(x_i))$ with discount factor $0 < \gamma < 1$. This definition of a “relative reward” is positive, when $f(x_{i-1}) > f(x_i)$. In this case, a_i leads to a smaller objective function value than a_{i-1} which means the more recent agent’s choice a_i

is closer to the location of the minimum than the previous action a_{i-1}. So this reward definition rewards agents which converge to the minimum as desired. It is worth mentioning: The smaller the objective function value $f(x_i)$ is, the greater the reward r_i. In addition to that, the term γ^{i-1} converges to zero for higher i. This means that high rewards achieved late in an episode are less worth than high rewards at the beginning of an episode. Hence, agents which converge faster to small objective function values get higher overall rewards.

Analogously, we can define a notion we call “relative environment”. We specified an action a in the environment to represent absolute input coordinates. This means, in the case of two optimization parameters, the action $a = (x_1, x_2)$ results in evaluating the objective function f at the input (x_1, x_2) independent of the previuos input. However, there is another viable option. One could define an action a to represent a stepvector. Assume the last evaluated input was $(x1_{\text{old}}, x2_{\text{old}})$ and the policy proposes the action $a = (\Delta x_1, \Delta x_2)$. Instead of evaluating the objective function f on $(\Delta x_1, \Delta x_2)$, we would use the input $(x1_{\text{old}}, x2_{\text{old}}) + (\Delta x_1, \Delta x_2) = (x1_{\text{old}} + \Delta x_1, x2_{\text{old}} + \Delta x_2) = (x1_{\text{new}}, x2_{\text{new}})$. When implementing “relative environments” one has to deal with two important details. First, adding the old input to the new action could exceed the input domain. One solution would be to map values greater than the maximum of the domain to the maximum value. Analogously, one could proceed for too small values. Second, one has to decide on a maximum step size. The step size is defined by the action domain. A step size which is too small could lead to slow convergence, even with a sophisticated optimization algorithm. On the other hand, the previously described exceeding of the domain boundaries is not desired. If the step size is too big, this might encourage this behavior.

Moreover, we already included implementations of the derivatives of all used objective functions in our code, but we did not include first-order information in observations. Including this additional information seems promising since the more information the observation contains, the faster and the more accurate the agent might learn. However, this would entail slightly higher memory consumption.

In this work, we used feedforward neural networks and recurrent neural networks for our experiments. Stochastic neural networks, which were inspired by Sherrington-Kirkpatrick models, are another class of artificial neural networks created by adding random fluctuations to the artificial neurons of the network, by using stochastic alternatives for the transfer functions or the weights of the artificial network. Thus, they could be effective for solving optimization tasks because the network could overcome local minima due to the random variations.[Tur04]

7 Conclusion

In this thesis, we proposed an approach for learning optimization algorithms for functions including parameter-dependencies and up to ten input dimensions. We defined it as a reinforcement learning problem, where every optimization algorithm can be viewed as a policy. For the evaluation, we mainly inspected a normalized version of the training reward and the mean squared error between the predicted minima and the actual ones. We implemented eight different objective functions to verify our approach. To suit our objective, these are scalable in dimension. All experiments show that the normalized training reward increases and the mean squared error decreases with higher training iteration steps. We compared how the number of free parameters affects the “learning” process for fixed tuples of input dimension and objective function. The more free parameters we have, the better the optimization algorithm performs for a fixed input dimension. Better performing optimization algorithms are characterized by higher rewards and lower mean squared errors. Furthermore, we analyzed the minimal and maximal number of free parameters per objective function for variable input dimensions. Smaller input dimensions lead to higher rewards and lower mean squared errors for one “free” parameter. The results for the maximum number of “free” parameters are dependent on the underlying objective function. After that, we investigated the influence of higher numbers of training iteration steps. We have shown with one example that this can have a major impact on the normalized training reward as well as on the mean squared error. Additionally, we explored the differences between using the REINFORCE- and the PPO-agent. PPO-agents achieve higher rewards and lower mean squared errors for low numbers of training iteration steps, but REINFORCE-agents outperform them in the long-term. Using the PPO-agent causes the runtime to grow drastically. Moreover, we utilized recurrent neural networks with long short-term memories to study if this boosts the performance of “learned” optimizers in comparison to using feedforward neural networks. It turned out that feedforward neural networks are sufficient for our use case and ensure lower runtime. Finally, we discussed the runtimes of the various experiments we conducted. The higher the input dimension, the higher the runtime. However, the number of free parameters has no influence on the runtime. The proposed approach could find application in areas operating with many “free” parameters, like it is the case in post-silicon validation.

Bibliography

[ADG+16] M. Andrychowicz, M. Denil, S. Gomez, M.W. Hoffman, D. Pfau, T. Schaul, B. Shillingford, N. De Freitas. “Learning to learn by gradient descent by gradient descent”. In: *Advances in neural information processing systems* 29 (2016) (cit. on pp. 9, 11).

[Bel57] R. Bellman. “A Markovian decision process”. In: *Journal of mathematics and mechanics* (1957), pp. 679–684 (cit. on p. 15).

[DHS11] J. Duchi, E. Hazan, Y. Singer. “Adaptive subgradient methods for online learning and stochastic optimization.” In: *Journal of machine learning research* 12.7 (2011) (cit. on p. 11).

[Doz16] T. Dozat. “Incorporating nesterov momentum into adam”. In: (2016) (cit. on p. 11).

[DPRL21] P. Domanski, D. Plüger, J. Rivoir, R. Latty. “Self-learning tuning for post-silicon validation”. In: *arXiv preprint arXiv:2111.08995* (2021) (cit. on pp. 10, 12).

[DSC+16] Y. Duan, J. Schulman, X. Chen, P.L. Bartlett, I. Sutskever, P. Abbeel. “Rl^2: Fast reinforcement learning via slow reinforcement learning”. In: *arXiv preprint arXiv:1611.02779* (2016) (cit. on p. 12).

[EKKP18] A. Eremeev, M. Khachay, Y. Kochetov, P. Pardalos. *Optimization problems and their applications*. Springer, 2018 (cit. on p. 9).

[FHM18] S. Fujimoto, H. Hoof, D. Meger. “Addressing function approximation error in actor-critic methods”. In: *International conference on machine learning*. PMLR. 2018, pp. 1587–1596 (cit. on p. 27).

[GSC00] F. A. Gers, J. Schmidhuber, F. Cummins. “Learning to forget: Continual prediction with LSTM”. In: *Neural computation* 12.10 (2000), pp. 2451–2471 (cit. on p. 18).

[GWR+16] A. Graves, G. Wayne, M. Reynolds, T. Harley, I. Danihelka, A. Grabska-Barwińska, S. G. Colmenarejo, E. Grefenstette, T. Ramalho, J. Agapiou, et al. “Hybrid computing using a neural network with dynamic external memory”. In: *Nature* 538.7626 (2016), pp. 471–476 (cit. on p. 12).

[HMW+20] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, T. E. Oliphant. “Array programming with NumPy”. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362 (cit. on p. 31).

[HS96] S. Hochreiter, J. Schmidhuber. “LSTM can solve hard long time lag problems”. In: *Advances in neural information processing systems* 9 (1996) (cit. on p. 18).

[HS97] S. Hochreiter, J. Schmidhuber. “Long short-term memory”. In: *Neural computation* 9.8 (1997), pp. 1735–1780 (cit. on p. 11).

[Hun07] J. D. Hunter. “Matplotlib: A 2D graphics environment”. In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95 (cit. on p. 31).

[HZAL18] T. Haarnoja, A. Zhou, P. Abbeel, S. Levine. “Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor”. In: *International conference on machine learning*. PMLR. 2018, pp. 1861–1870 (cit. on p. 27).

[KB14] D. P. Kingma, J. Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014) (cit. on p. 11).

[LHP+15] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, D. Wierstra. “Continuous control with deep reinforcement learning”. In: *arXiv preprint arXiv:1509.02971* (2015) (cit. on p. 27).

[Li17] K. Li. *Learning to Optimize with Reinforcement Learning*. Accessed: 2022-09-09. 2017. URL: %5Curl%7Bhttps://bair.berkeley.edu/blog/2017/09/12/learning-to-optimize-with-rl/%7D (cit. on p. 20).

[LM16] K. Li, J. Malik. “Learning to optimize”. In: *arXiv preprint arXiv:1606.01885* (2016) (cit. on pp. 9, 11).

[Lug05] G. F. Luger. *Artificial intelligence: structures and strategies for complex problem solving*. Pearson education, 2005 (cit. on p. 14).

[MAP+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhouc, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/ (cit. on p. 30).

[MBD+90] T. Mitchell, B. Buchanan, G. DeJong, T. Dietterich, P. Rosenbloom, A. Waibel. “Machine learning”. In: *Annual review of computer science* 4.1 (1990), pp. 417–433 (cit. on p. 14).

[MKS+13] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmiller. “Playing atari with deep reinforcement learning”. In: *arXiv preprint arXiv:1312.5602* (2013) (cit. on p. 27).

[MS05] M. Molga, C. Smutnicki. “Test functions for optimization needs”. In: *Test functions for optimization needs* 101 (2005), p. 48 (cit. on p. 22).

[OW12] M. v. Otterlo, M. Wiering. “Reinforcement learning and markov decision processes”. In: *Reinforcement learning*. Springer, 2012, pp. 3–42 (cit. on p. 16).

[PGM+19] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, S. Chintala. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf (cit. on p. 30).

[Pic94] P. Picton. "What is a neural network?" In: *Introduction to Neural Networks*. Springer, 1994, pp. 1–12 (cit. on p. 16).

[PMB13] R. Pascanu, T. Mikolov, Y. Bengio. "On the difficulty of training recurrent neural networks". In: *International conference on machine learning*. PMLR. 2013, pp. 1310–1318 (cit. on p. 17).

[PWG13] V. Picheny, T. Wagner, D. Ginsbourger. "A benchmark of kriging-based infill criteria for noisy optimization". In: *Structural and multidisciplinary optimization* 48.3 (2013), pp. 607–626 (cit. on pp. 23, 62).

[SB18] R. S. Sutton, A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018 (cit. on p. 14).

[SC16] Y. C. M. W. H. Sergio, G. Colmenarejo. "Learning to Learn for Global Optimization of Black Box Functions". In: *stat* 1050 (2016), p. 18 (cit. on pp. 9, 12).

[Sch22] K. Schüttler. *Investigation of self-learned zeroth-order optimization algorithms*. Bachelor's Thesis. 2022 (cit. on p. 53).

[SWD+17] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, O. Klimov. "Proximal policy optimization algorithms". In: *arXiv preprint arXiv:1707.06347* (2017) (cit. on p. 27).

[Tur04] C. Turchetti. *Stochastic models of neural networks*. Vol. 102. IOS Press, 2004 (cit. on p. 54).

[Wil92] R. J. Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine learning* 8.3 (1992), pp. 229–256 (cit. on p. 27).

[Zei12] M. D. Zeiler. "Adadelta: an adaptive learning rate method". In: *arXiv preprint arXiv:1212.5701* (2012) (cit. on p. 11).

All links were last followed on November 2, 2022.

A Gradients of Objective Functions

In this chapter, we list gradients of all the objective functions we used to train our optimization algorithm in alphabetic order. The gradient for a function f of dimension d with input $x = (x_1, \ldots, x_d)$ is defined as follows:

$$\text{(A.1)}\quad \nabla f(x) = \nabla f(x_1, x_2, \ldots, x_d) = \begin{bmatrix} \dfrac{\partial f}{\partial x_1}(x_1, x_2, \ldots, x_d) \\ \dfrac{\partial f}{\partial x_2}(x_1, x_2, \ldots, x_d) \\ \dfrac{\partial f}{\partial x_n}(x_1, x_2, \ldots, x_d) \end{bmatrix}$$

We define $\forall i \in \{1, \ldots, d\} : \nabla f(x)_i = \dfrac{\partial f}{\partial x_i}(x)$ and specify gradients of objective functions by specifying $\nabla f(x)_i$ for all possible i.

Ackley Function

The Ackley function is almost flat in its outer region and its global minimum lies in a large central hole. The outer region contains multiple evenly distributed local minima, in which optimization algorithms might get trapped in. We used $a = 20, b = 0.2$ and $c = 2\pi$

$$\text{(A.2)}\quad \nabla f(x)_i = \frac{ab \cdot x_i}{\sqrt{d \cdot \sum_{j=1}^{d} x_j^2}} \cdot exp\left(-b \cdot \sqrt{\frac{1}{d} \cdot \sum_{j=1}^{d} x_j^2}\right) + \frac{c}{d} \cdot \sin(c \cdot x_i) \cdot exp\left(\frac{1}{d} \cdot \sum_{j=1}^{d} \cos(c \cdot x_j)\right) \qquad \forall i \in \{1, \ldots, d\}$$

Griewank Function

The Griewank function owns a lot of local minima, spread over the entire domain. The surface area of the Griewank function has a very fine texture.

$$\text{(A.3)}\quad \nabla f(x)_i = \frac{1}{2000} \cdot x_i + \frac{1}{\sqrt{i}} \cdot \sin\left(\frac{1}{\sqrt{i}} \cdot x_i\right) \qquad \forall i \in \{1, \ldots, d\}$$

Levy Function

The Levy function has the most complex gradient of our functions. Let $\forall i \in \{1, \ldots, d\} : w_i = 1 + \frac{x_i - 1}{4}$. Then the function and its gradient can be defined as:

$$(A.4)\quad \nabla f(x)_i = \begin{cases} \frac{\pi}{2}\sin(\pi w_i)\cos(\pi w_i) \\ \quad +\frac{1}{2}(w_i - 1)(1 + 10\sin^2(\pi w_i + 1)) \\ \quad +5\pi(w_i - 1)^2 \sin(\pi w_i + 1)\cos(\pi w_i + 1) & \text{, if } i = 1 \\ \frac{1}{2}(w_i - 1)(1 + 10\sin^2(\pi w_i + 1)) \\ \quad +5\pi(w_i - 1)^2 \sin(\pi w_i + 1)\cos(\pi w_i + 1) & \text{, if } i \in \{2, \ldots, d-1\} \\ \frac{1}{2}(w_i - 1)(1 + \sin^2(2\pi w_i)) \\ \quad +\pi(w_i - 1)^2 \sin(2\pi w_i)\cos(2\pi w_i) & \text{, if } i = d \end{cases}$$

Rastrigin Function

The Rastrigin function owns multiple regularly distributed local minima.

$$(A.5)\quad \nabla f(x)_i = 2 \cdot x_i + 20\pi \cdot \sin(2\pi \cdot x_i) \qquad \forall i \in \{1, \ldots, d\}$$

Rosenbrock Function

The Rosenbrock function is a member of the valley-shaped functions. It is also called Rosenbrock's valley or Rosenbrock's banana function. The global minimum lies in a narrow, oblong, parabolic-shaped valley. Optimization Algorithms tend to find this valley rather quickly, but struggle to find the global minimum inside. [PWG13]

$$(A.6)\quad \nabla f(x)_i = \begin{cases} -400 \cdot x_i \cdot (x_{i+1} - x_i^2) + 2(x_i - 1) & \text{, if } i = 1 \\ -400 \cdot x_i \cdot (x_{i+1} - x_i^2) + 2 \cdot (x_i - 1) + 200 \cdot (x_i - x_{i-1}^2) & \text{, if } i \in \{2, \ldots, d-1\} \\ 200 \cdot (x_i - x_{i-1}^2) & \text{, if } i = d \end{cases}$$

Sphere Function

The Sphere function is the simplest among our objective functions. This bowl-shaped function has no minimum except for the one global minimum in the center of its domain. Moreover, this is the only strictly convex function we used.

$$(A.7)\quad \nabla f(x)_i = 2 \cdot x_i \qquad \forall i \in \{1, \ldots, d\}$$

Styblinski-Tang Function

The Styblinski-Tang function has a unique curvy bowl shape.

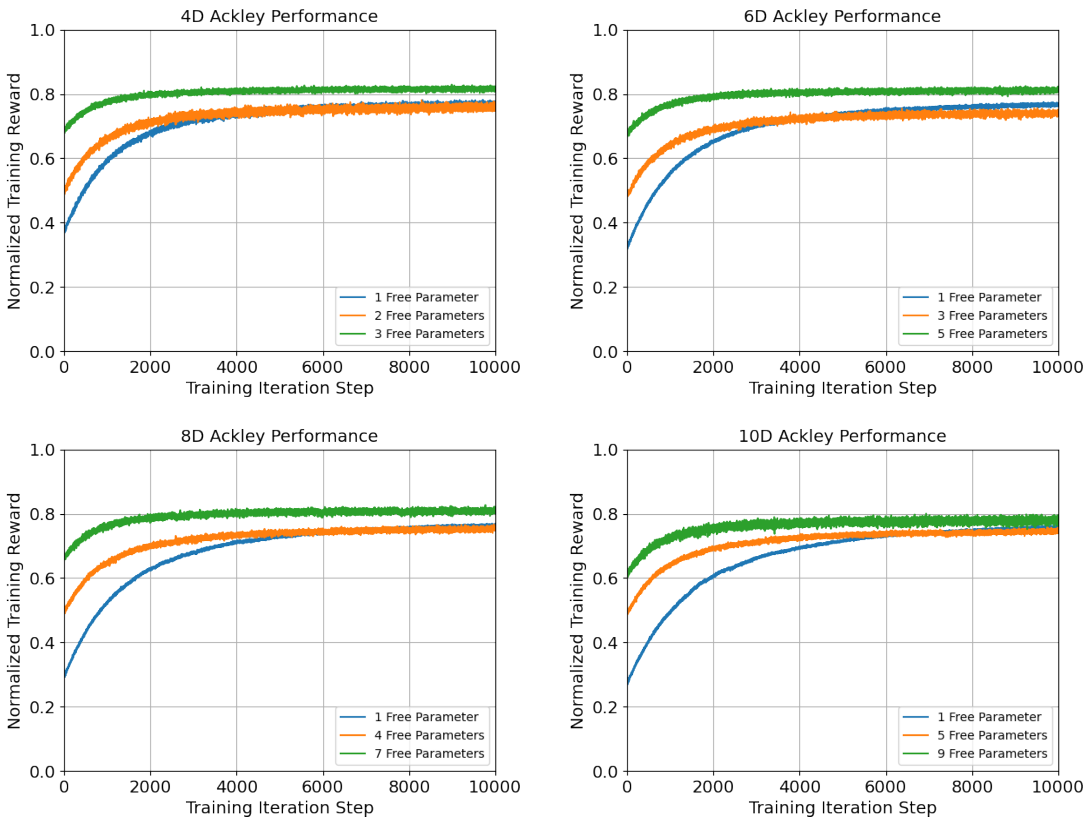
$$\text{(A.8)}\quad \nabla f(x)_i = \frac{1}{2} \cdot (4x_i^3 - 32x_i + 5) \qquad \forall i \in \{1, \dots, d\}$$

Zakharov Function

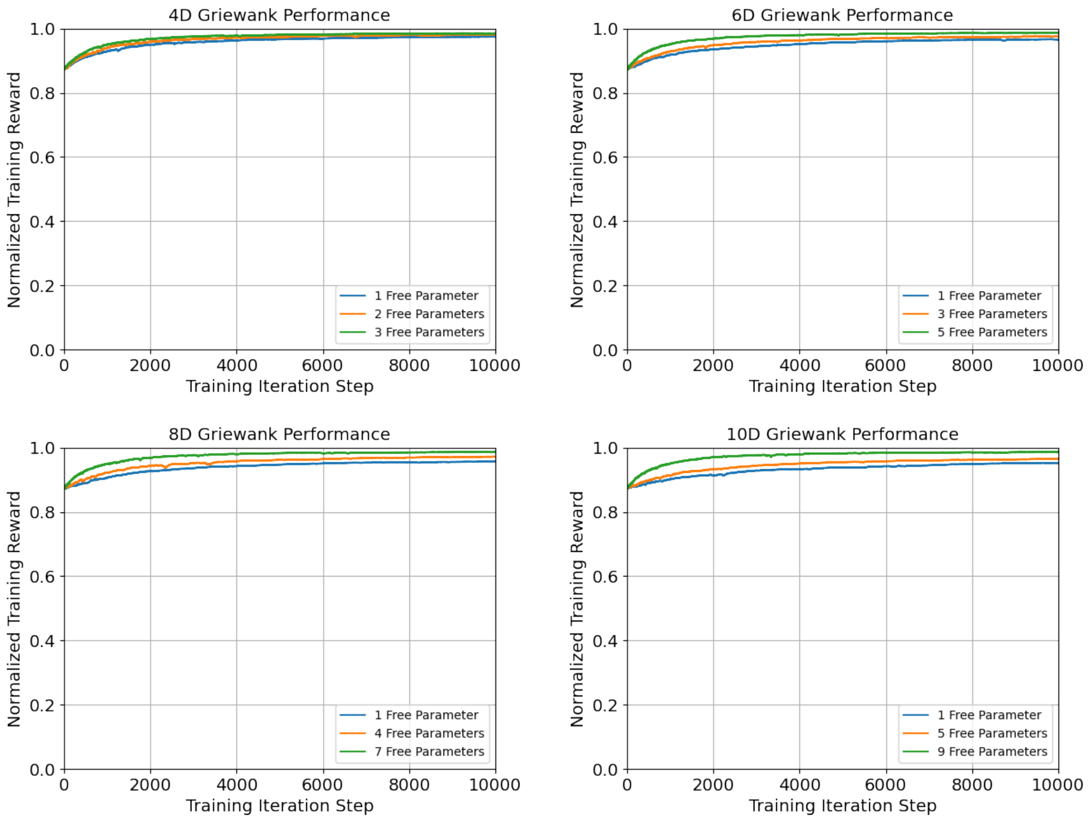
The Zakharov function is the only representative of plate-shaped functions in our collection. It has no local minimum besides the global one.

$$\text{(A.9)}\quad \nabla f(x)_i = 2x_i + \frac{1}{2} i \cdot \sum_{j=1}^{d} (j x_j) + 2i \cdot \left(\frac{1}{2} \cdot \sum_{j=1}^{d} j \cdot x_j \right)^3 \qquad \forall i \in \{1, \dots, d\}$$

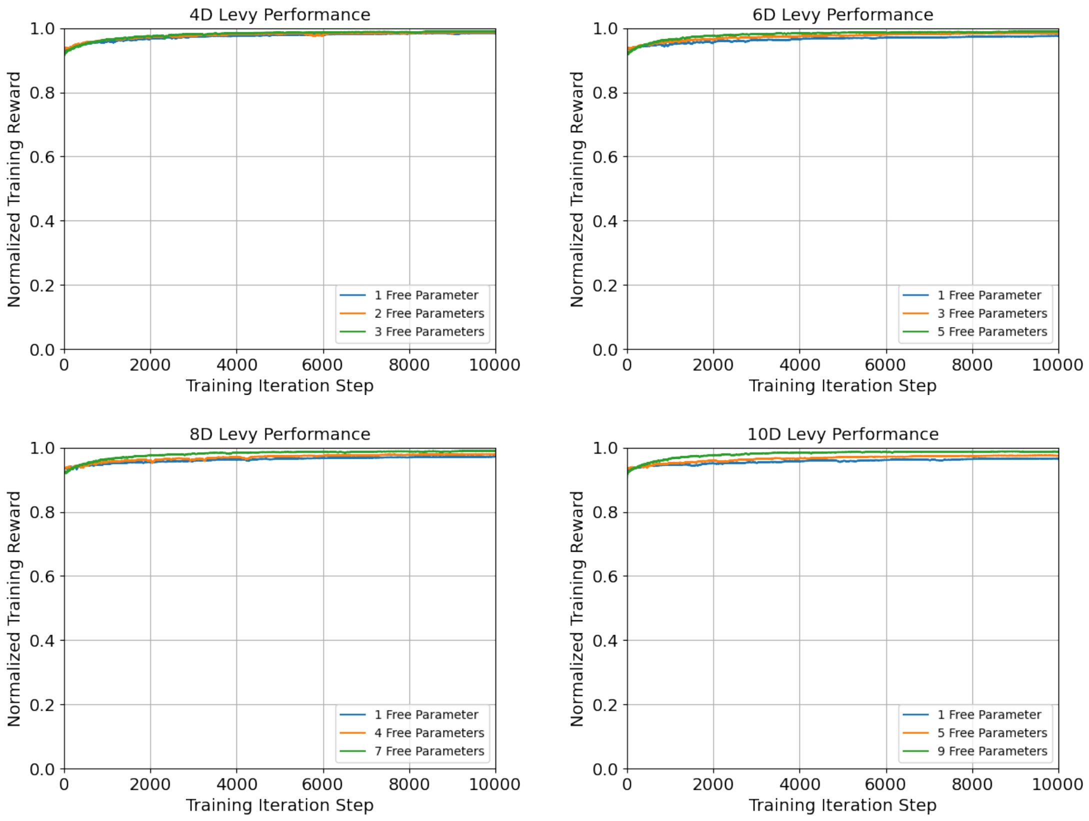
B Complete Experimental Results



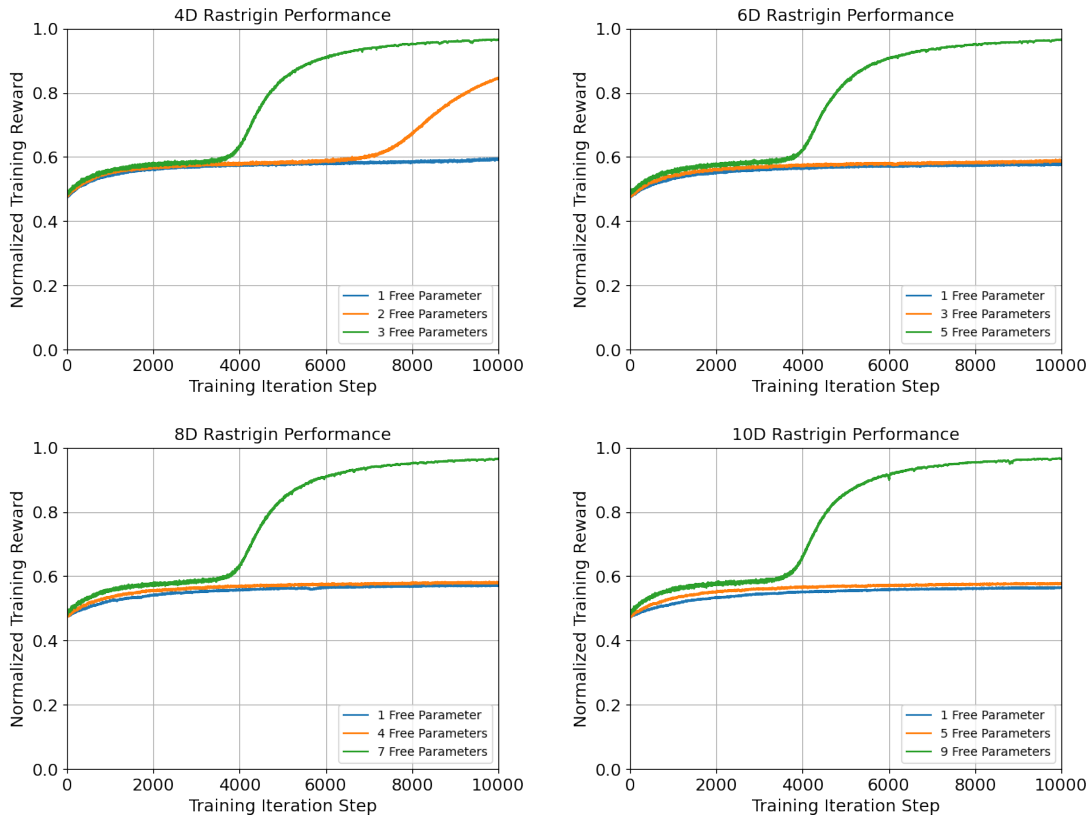
(a) Ackley



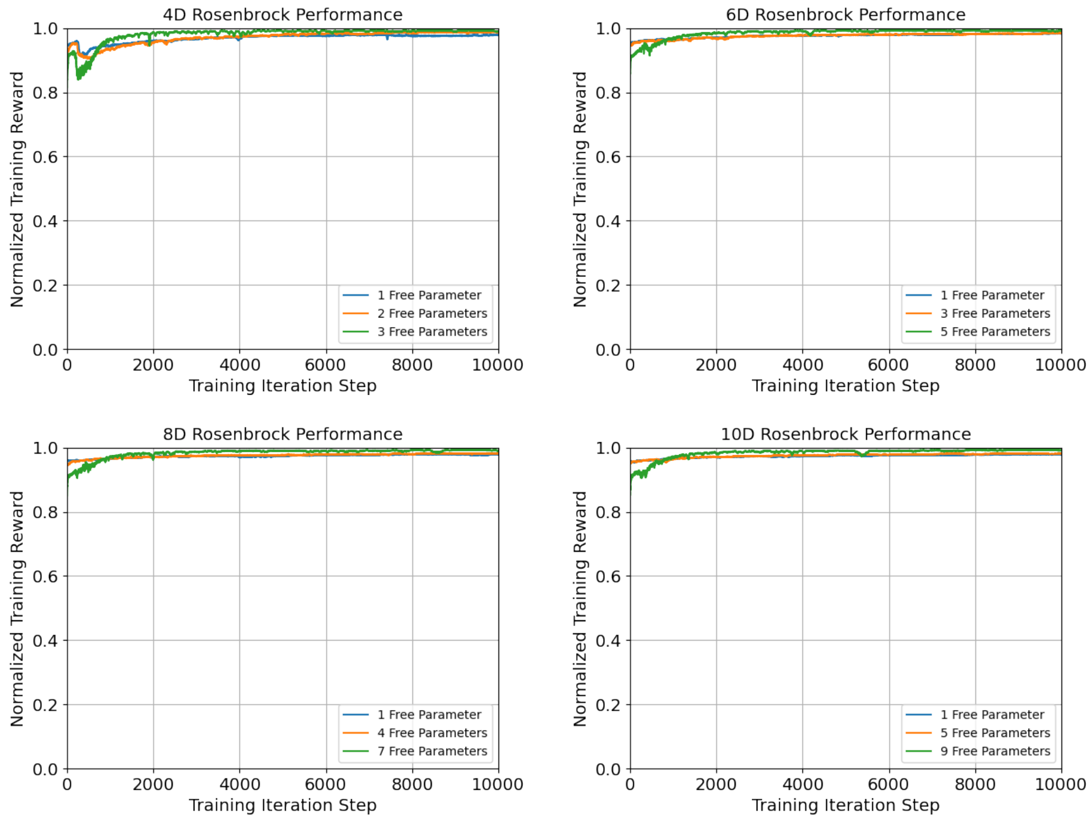
(b) Griewank



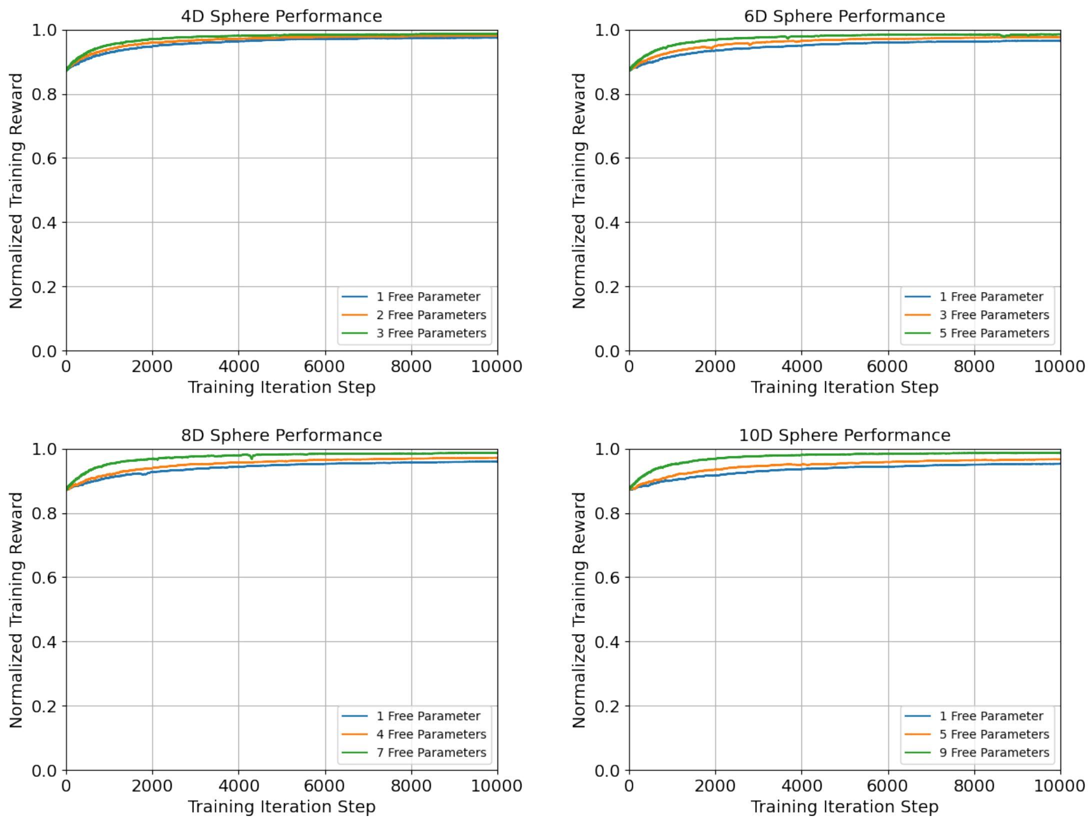
(c) Levy



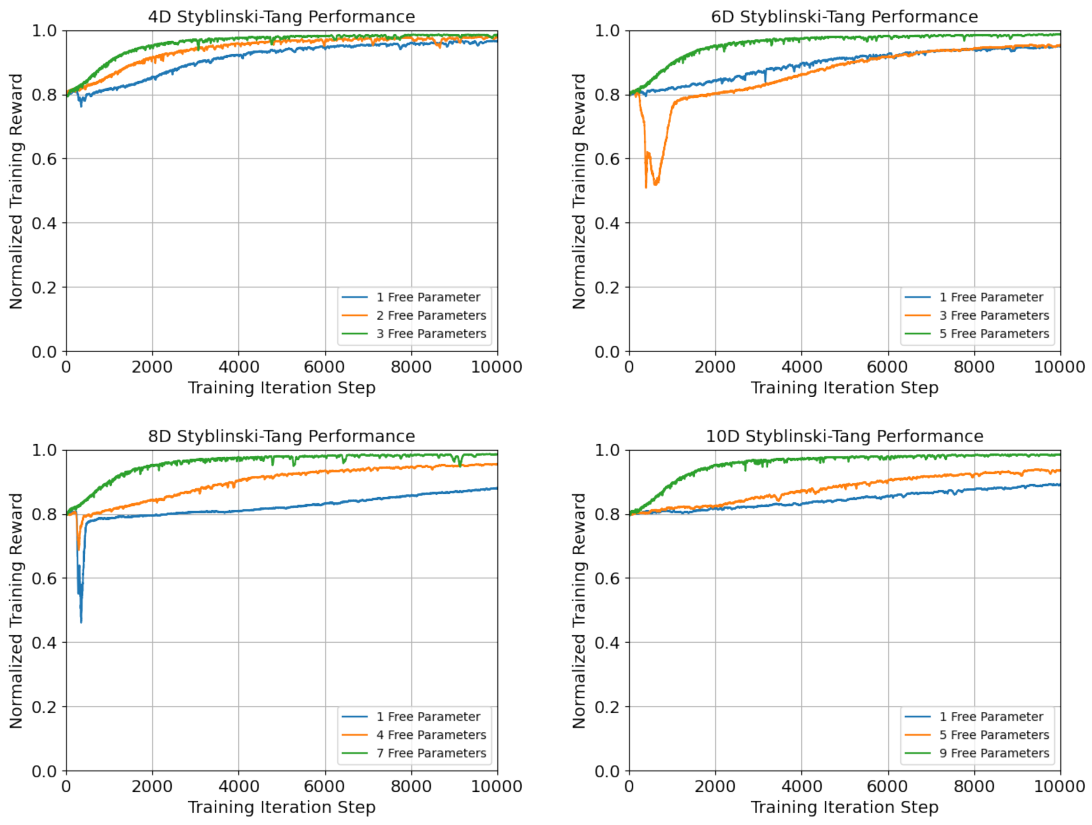
(d) Rastrigin



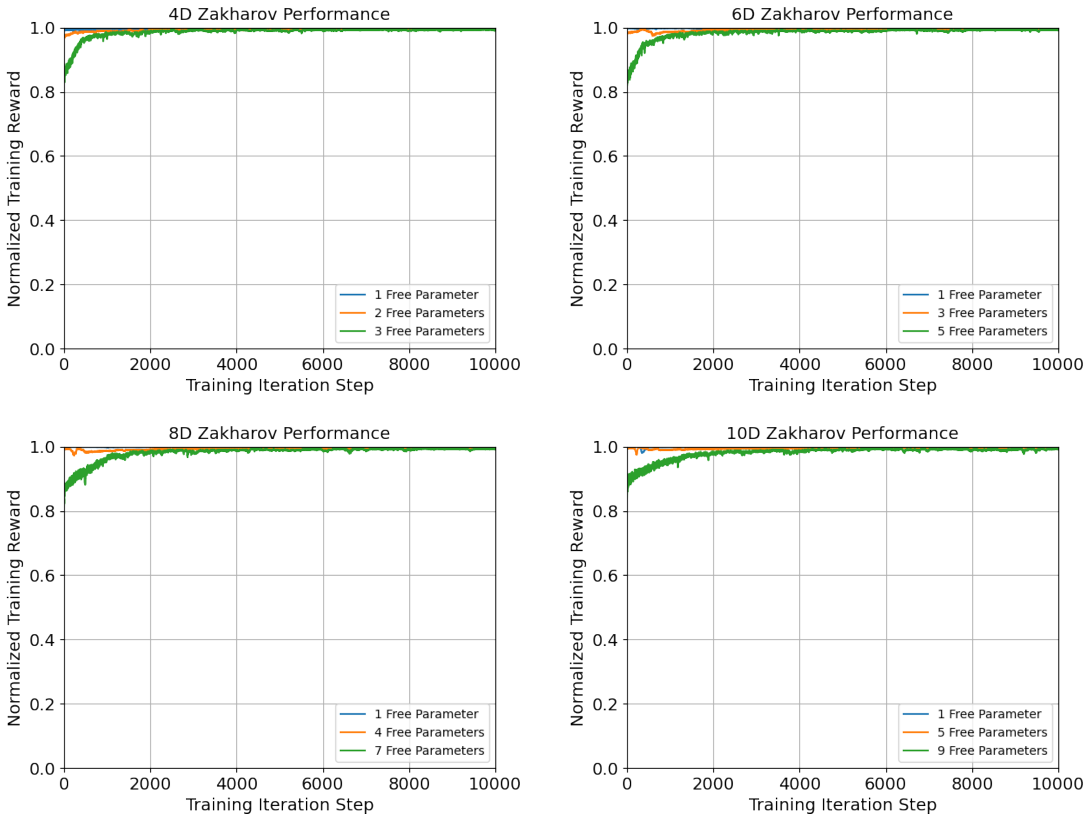
(e) Rosenbrock



(f) Sphere

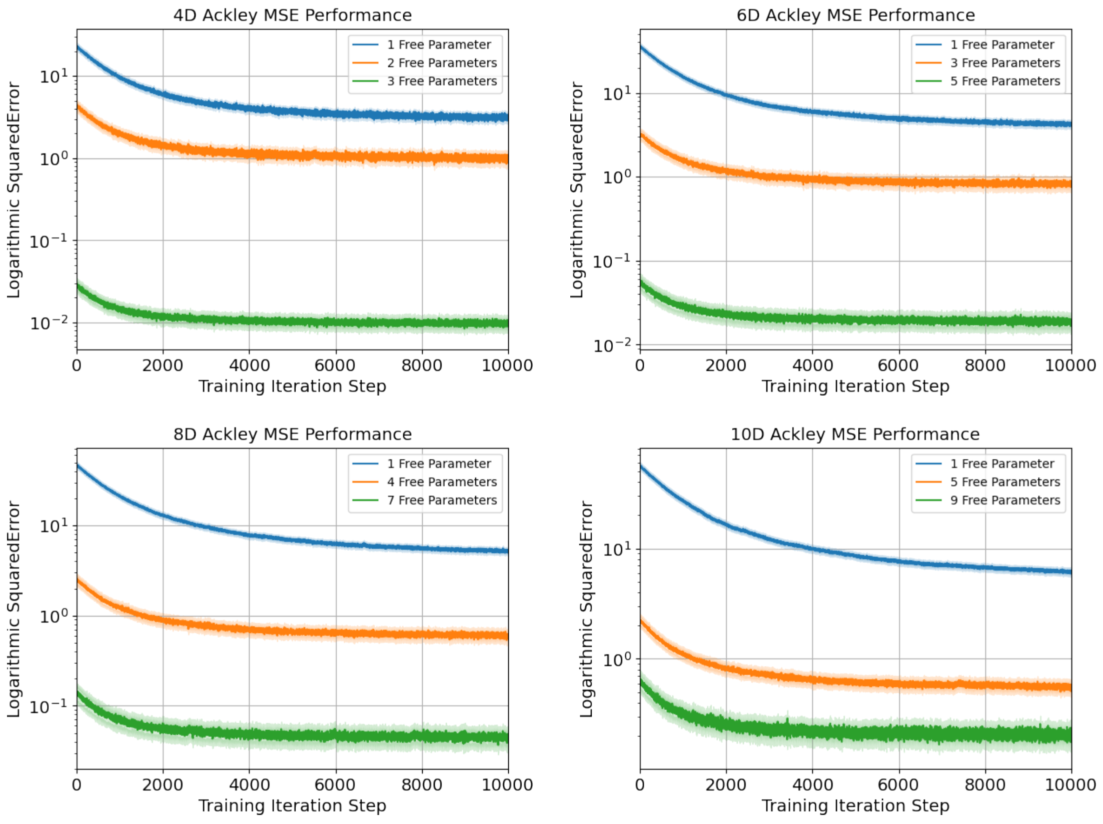


(g) Styblinski-Tang

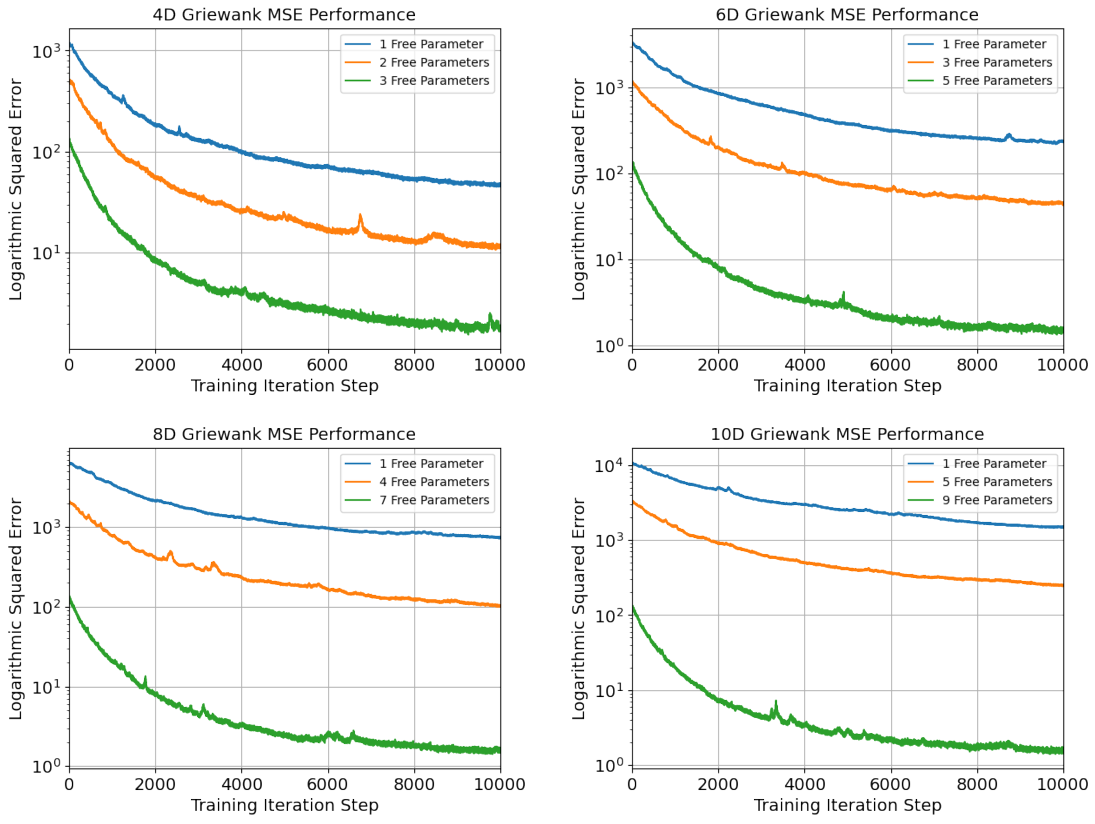


(h) Zakharov

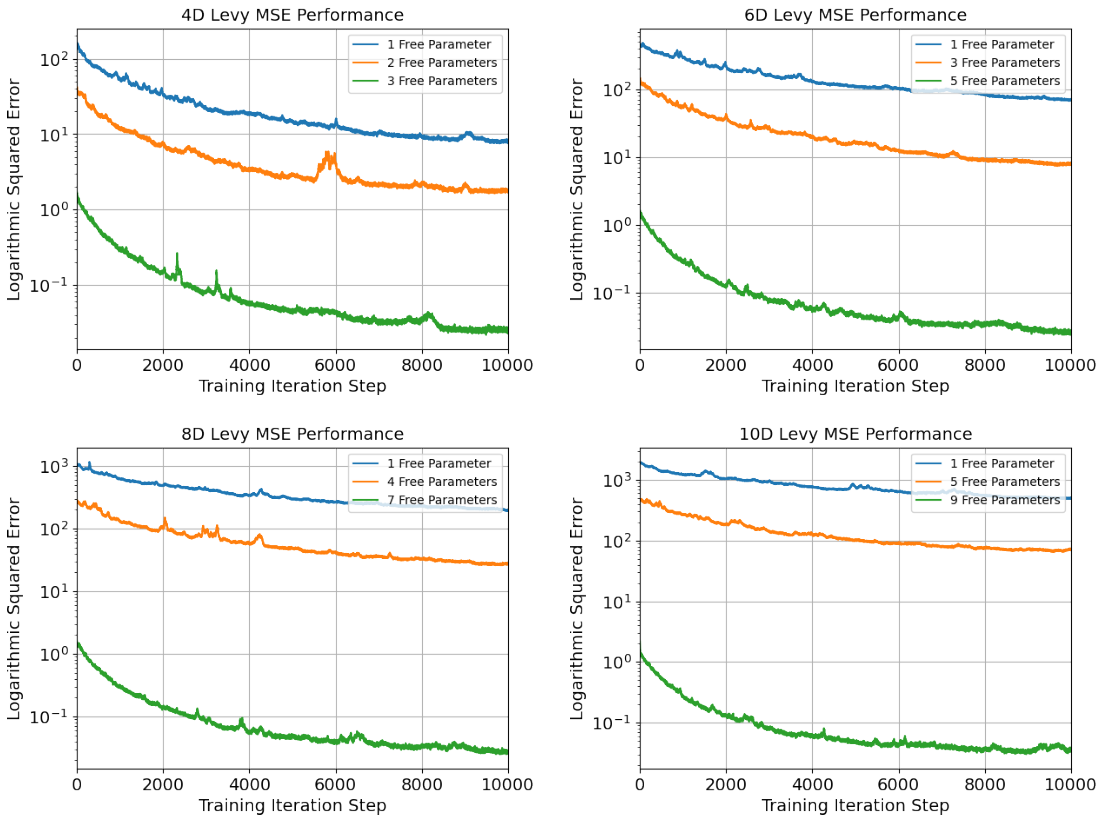
Figure B.1: Parameter-dependent performance for all input dimensions on the eight objective functions



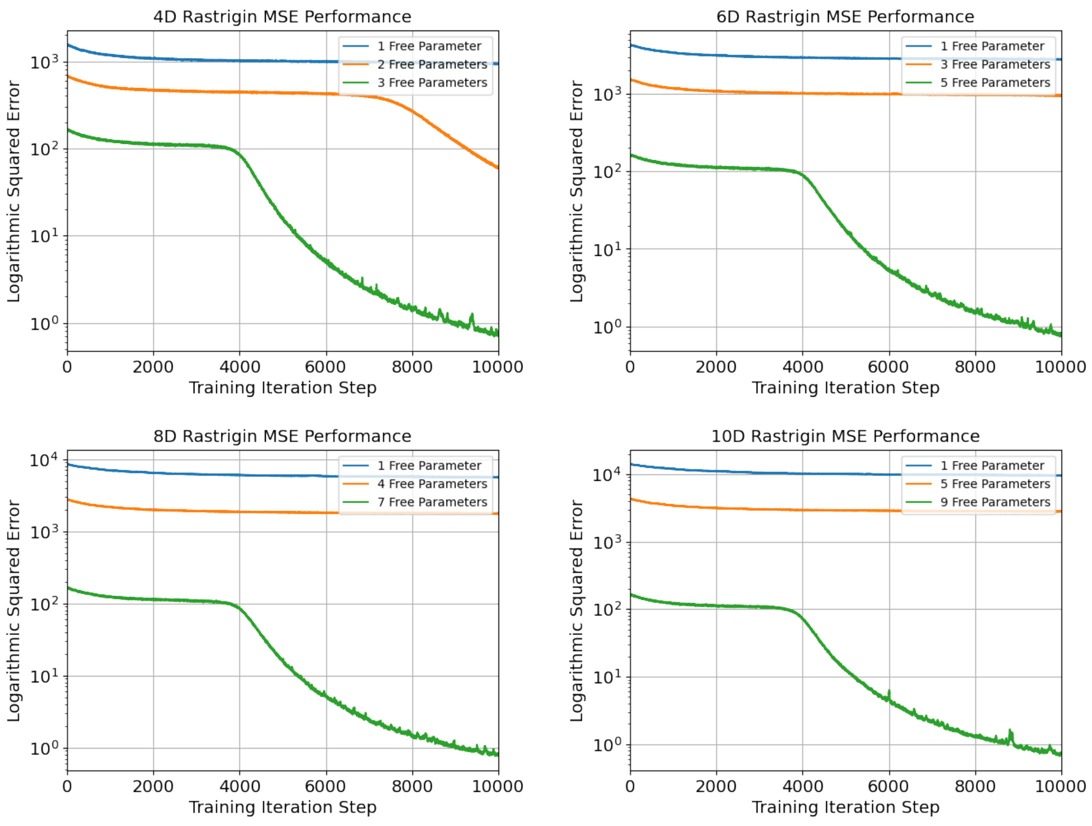
(a) Ackley



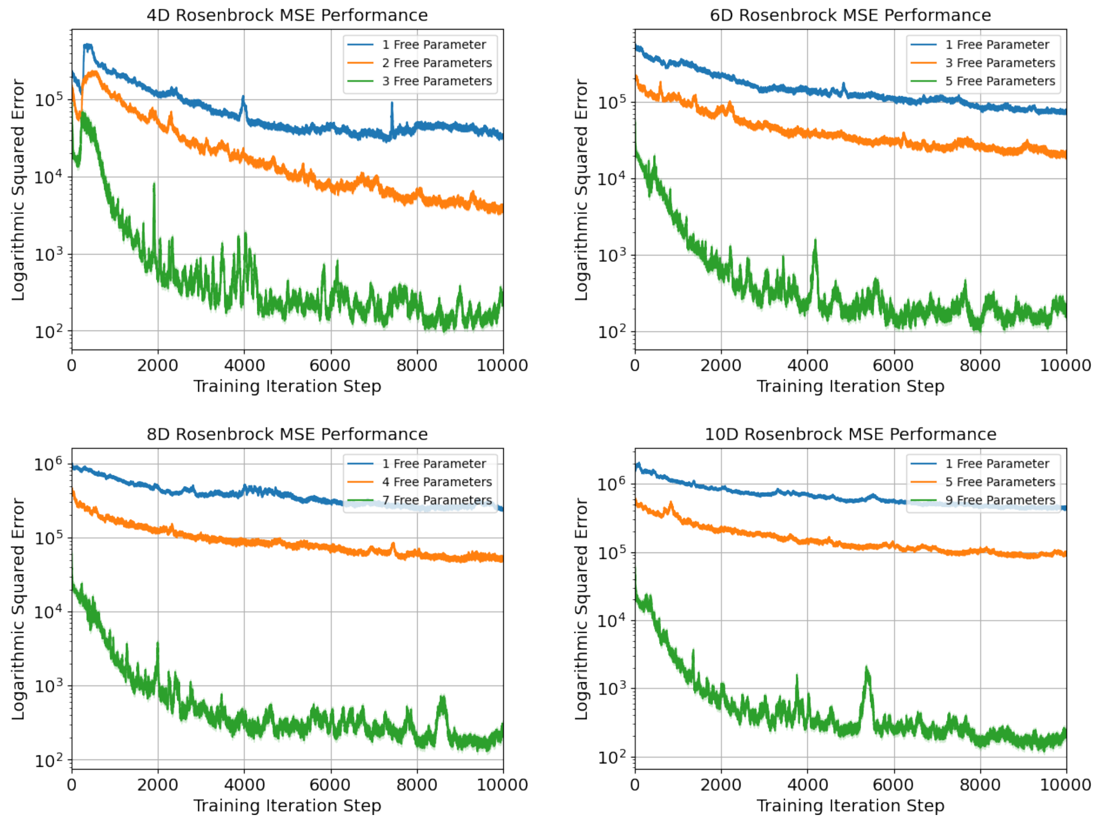
(b) Griewank



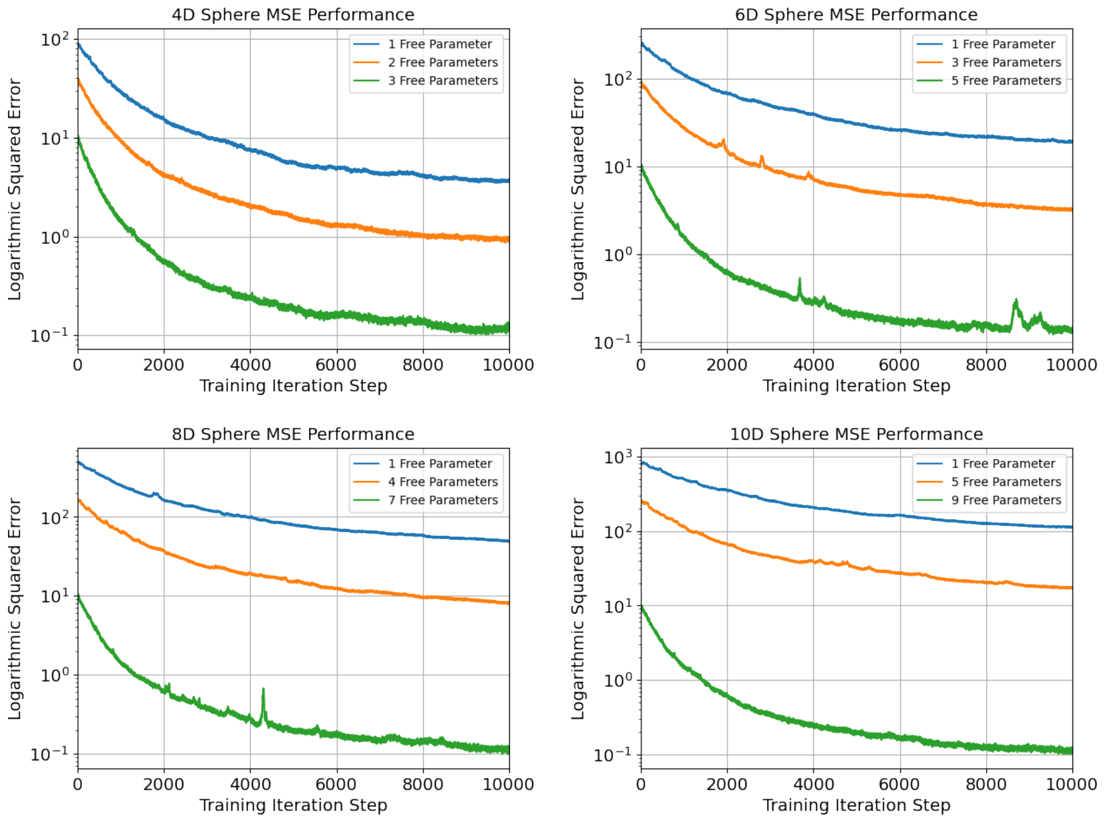
(c) Levy



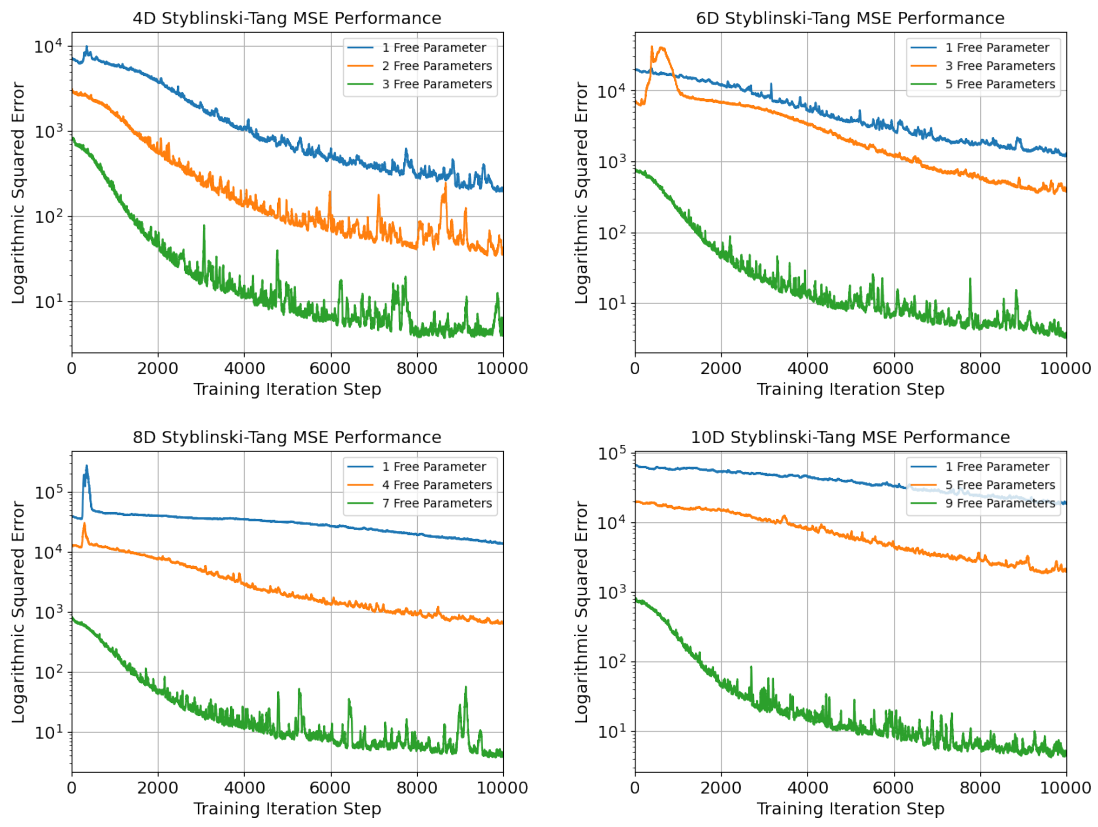
(d) Rastrigin



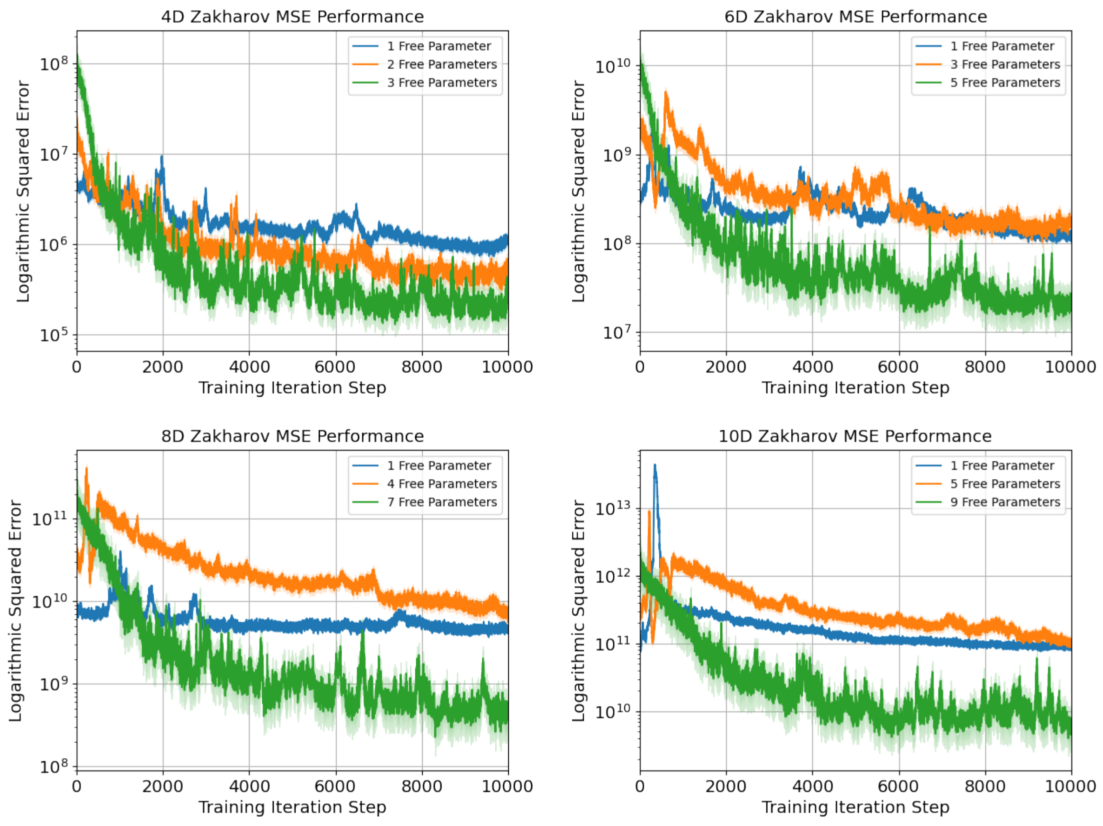
(e) Rosenbrock



(f) Sphere



(g) Styblinski-Tang



(h) Zakharov

Figure B.2: Parameter-dependent MSE for all input dimensions on the eight objective functions

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature